From FPGA to ASIC: A RISC-V processor experience

Carlos Rojas Morales

Advisors:
Francesc Moll Echeto
Adrián Cristal Kestelman
Marco Antonio Ramírez Salinas

Barcelona School of Informatics - Universitat Politècnica de Catalunya
Computing Research Center - Instituto Politécnico Nacional

A thesis submitted in fulfillment of the requirements for the degree of
Master of Science

October 18, 2019

Abstract

RISC-V has become more relevant in the computer architecture research field, in this context, a correct methodology for using electronic design automation (EDA) tools for verification and Synthesis becomes crucial to generate a tape-out for silicon manufacturing. The main objective of this work is to determine and document a correct design flow using these tools in the Lagarto RISC-V Processor and the RTL design considerations that must be taken into account, to move from a design for FPGA to design for ASIC.

Acknowledgements

This thesis work was a joint effort to achieve the objective of developing the first tape-out attempt of a RISC-V microprocessor by the involved institutions. I want to thank all the members of the DRAC team for their tremendous effort to reach the deadline dates and every contribution that makes possible the successful accomplishment of this project.

I express my gratitude to my advisors Francesc Moll, Adrián Cristal, and Marco Antonio Ramírez. Without your support, this thesis work would not have been possible.

I want to express my gratitude to the professors of the CIC-IPN and the UPC for the solid knowledge provided to me.

And of course, to my family and friends that always listen to me, support me, and encouraged me to keep going in the difficult times. I want to mention, especially to my parents, German and Mariana, and my girlfriend Daniela.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The semiconductor industry for High-Performance Computing is a very competitive market that has been dominated by a few companies over the years. Most industrial aspects of computer architecture, like the Application Specific Integrated Circuit (ASIC) deployment, have been being delegated to these few companies, and the stronger research efforts have been made mostly at the highest levels of abstraction. In the last years, the interests in investigation and development around the design and fabrication of open-source microprocessors have been increased [1–4]. News, like the Huawei ban by the US Government due to security issues, motivates the necessity of having more control and independence in technology development. Open-source Hardware (HW) and Software (SW) projects aim to give more control and transparency not only from the SW part of an application, instead they bring the advantages of the open-source to the HW design and implementation. With this approach, the joint development of open-source HW and SW will reach more levels of security, and also it will bring more technological independence.

In this context, the European Commission announces the selection of the *Consortium European Processor Initiative (EPI)* to co-design, develop, and bring on the market a European low-power microprocessor [5]. The *Barcelona Supercomputing Center (BSC)*, as part of this consortium, have been started developing the first versions of custom silicon chips for High Performance Computing (HPC), including HW accelerators and SW stack for the exascale and pre-exascale computing targets.

One of these projects is the *DRAC* project. *DRAC* is an international collaboration involving multiple institutions, including, the *BSC*, *Centre Na-*

cional de Microelectrònica (CNM), *Centro de Investigación en Computación del Instituto Politécnico Nacional (CIC-IPN)* and *Universitat Politècnica de Catalunya (UPC)*. *DRAC* is an internal BSC project, and it is partially supported by the Spanish Ministry of Science and Technology (project TIN2015-65316-P) and by the Generalitat de Catalunya (2017-SGR-1328). The initial phase of this project consists of the design, verification, implementation, and fabrication of a RISC-V general-purpose microprocessor capable of booting Linux. This design will be the first in a series of HPC microprocessors designed in the context of the *EPI*. The product of this initial phase will be a functional ASIC for a SoC. This SoC will use a test board to have a main memory communication, a JTAG communication port, a UART controller, and an SPI controller to access an SD card. This initial RISC-V microprocessor and SoC implementation will be the basis for the designs that will be developed in *DRAC* and, for this reason it is denoted as *preDRAC* SoC.

On the other hand, *Lagarto* is a platform that embraces a complete modular microprocessor architecture that was developed using Verilog-HDL. Nowadays, this platform is used to teach the computer Architecture Course in the *CIC-IPN*. *Lagarto* processor consists of a set of different modules working together to execute a full 32-bit Instruction Set Architecture (ISA), and it can execute integer and floating-point operations, it also counts with a Processor Local Bus (PLB) based on the *Wishbone* protocol and a Memory Management Unit (MMU), which includes two levels of hierarchy [6]. Currently, *Lagarto* is implemented at Register Transfer Level (RTL) using Verilog-HDL, and it is in the final verification phase. One goal of this platform is to fabricate it into a silicon chip.

The *Lagarto* microprocessor originally was implemented using the MIPS ISA, and it was modified to execute the 64-bit integer RISC-V ISA. This updated *Lagarto* was used as a base architecture of the *preDRAC* SoC. This microprocessor was used along with a modified *lowRISC* SoC [7] to perform the first Tape-out attempt of a complex system like a SoC, with peripherals, a memory hierarchy, the support of a complete ISA and the capability of booting a Linux kernel. Although RTL implementation and Field Programmable Gate Array (FPGA) prototyping are good approximations of how an RTL code behaves, once it has been fabricated, there are more considerations and missing steps before generating the final Graphics Data System II (GDSII) that should be sent to the foundry. These considerations include the die area, Input-Output (IO) available pins, physical interfaces, and debugging capabilities. Some essential steps before sending a tape-out to the foundry are verification, logic synthesis, Design For Testability (DFT), floor-planing, Place and Route (PnR), and Gate Level Simulation (GLS).

1.2 Objectives

1.2.1 General Objective

To modify and implement the design specifications of the *preDRAC* SoC to fabricate it, according to the physical specifications and constraints of the target fabrication technology.

1.2.2 Specific Objectives

- To define specifications considering the target technology, physical interfaces, testing features, and debugging features.
- To implement and to integrate at RTL level the modifications and new features to the *preDRAC* SoC.
- To develop a test environment for the SoC using FPGA prototyping.
- To adapt the RTL implementation to make it synthesizable.
- To set-up a synthesis environment to generate a net-list using Electronic Design Automation (EDA) tools and the target technology libraries.
- To set-up a GLS environment to perform post-synthesis verification.
- To define post-silicon validation tests.

1.3 Justification

Despite the growing development of the silicon industry over several years, nowadays, there are only a few companies surviving today that develop and fabricate complex systems in silicon, such as SoCs and microprocessors. This lack of companies is caused mainly because of Moore's Law [8], this law implied that the Integrated Circuit (IC) market has evolved at an exponential ratio. At the same time that the available transistors had been increasing, the system's complexity has increased. This complexity has forced the industry always to keep on the edge of technological capabilities. This constant improvement in the silicon chips fabrication technology keeps the cost of development and fabrication high, and consequently, the products are competitive only for a short time.

Many factors have contributed to increase the interest in research and develop projects related to the design and fabrication of IC. One of these

factors is the one mentioned in [9], "Moore's Law is already failing and is anticipated to flatten by 2025. Absent a new transistor technology to replace Complementary Metal-Oxide-Semiconductor (CMOS), the main opportunity for continued performance improvement for digital electronics and HPC is to make more effective use of transistors through more efficient architectures". This idea suggests that more research effort should be made to keep growing the development in the IC field. Also, the fact that Moore's Law is flattering could imply that more fabrication nodes will be accessible and will keep competitive for more time. Another important factor that motivates the research in the fabrication of silicon chips is the growing of the Internet of Things (IoT) market. The backbone of IoT is energetically autonomous wireless sensors [10]. These sensors are devices for ultra-low-power embedded applications. These devices require advanced signal processing capabilities to execute very heterogeneous tasks, e.g., managing the interfaces to acquire data, storing it into volatile/non-volatile memories, and transmitting the data via radio. The combination of heterogeneous capabilities and the constraints of area and power causes a growing interest for an extendable microprocessor's ISA [11]. The development of custom microprocessors for IoT devices has used the RISC-V open-source standard ISA [12]. Such as the microprocessors that are the presented in [3, 11]. The open-source characteristics of the RISC-V standard and the control capabilities of designing and fabricating a custom microarchitecture bring the opportunity of improving the security capabilities of a device by making sure that hidden back doors have not been introduced in the HW by an external partner.

In this context, projects like *EPI* enforce the motivation for the research and development in the fabrication of ASICs. The *preDRAC* project is a necessary step to obtain experience in the design and fabrication flow of ASICs that will lead the involved institutions to remain competitive in the computer architecture and micro-technology fields.

1.4 Related Work

In this section we will give a brief description of similar work related with RISC-V fabricated proccesors.

1.4.1 PULP

Riscy cores *Riscy* cores are a series of in order UltraLow-Power (ULP) cores. They are based in the RISC-V ISA and are intended to be applied in IoT devices. These cores are part of the Parallel Ultralow-Power Platform

(PULP), these RISC-V cores are embedded in a cluster to have very low power multi-core systems with extended Digital Signal Processing (DSP) capabilities [13]. There are three different implementations of *Riscy* core series, *Riscy*, *Zero-riscy* and *Micro-riscy*.

Riscy *Riscy* is an open-source 32b in-order core with four pipeline stages. Its RISC-V ISA implementation has been extended to enhance performance, reduce the code size, and increase the energy efficiency of signal processing algorithms. The core has been designed and optimized to work in a multi-core cluster.

Zero-riscy *Zero-riscy* is an area-optimized RISC-V core. It has two pipeline stages that are divided in *Instruction Fetch* and *Instruction Decode and Execute*. It contains a prefetch-buffer that collects data from the instruction memory. The Arithmetic Logic Unit (ALU) contains minimal hardware resources to implement the ISA: one 32b adder, one 32b shifter and the logic unit. *Zero-riscy* implements a minimum set of control-status register defined by the privileged RISC-V 1.9 spec.

Micro-riscy *Micro-riscy* is further optimized for area with respect to *Zero-riscy* by removing the RVM RISC-V extensions. *Micro-riscy* does not have any HW support for multiplications and divisions. To further reduce the area footprint, it implements the RVE RISC-V specification which allows to use only 16 general-purpose registers. The Riscy synthesis results are shown in Table 1.1

Quentin

On the other hand, there is the Quentin SoC, which is a single-core implementation of PULPissimo open-source platform. [1] This SoC were fabricated as well in a 22nm technology. This SoC features a 32-bit in-order 4-pipeline stages RISC-V processor. The baseline RISC-V ISA of the processor has been enhanced with extensions targeting energy-efficient digital signal processing. A remarkable feature is its heterogeneous memory system architecture, which is composed of a mix of Static Random-Access Memorys (SRAMs) and Standard-Cell Memory cuts (SCMs). This configuration provides the possibility of using an ultra-low mode power mode by implementing independent power sources to the SRAMs. Using only 16 kB of SCM memories and shutting down the SRAM via an off-chip power manager. The SoC

[1] https://github.com/pulp-platform/pulpissimo

Table 1.1: Riscy cores tape-out specifications.

Riscy	
Bits	32
RISC-V user spec	IMC
Technology	UMC 65nm
Target frequency	55 Mhz - 560 MHz
Riscy Area	40.7 KGE (58608 μm^2)*
Zero-riscy	40.7 KGE (58608 μm^2)*
Micro-riscy	40.7 KGE (58608 μm^2)*
Riscy Power	77 μW - 3.77 mW
Riscy IPC	0.79+

*Equivalent minimun-size NAND2 gate area.In UMC 65nm, one gate equivalent (GE) is 1.44 μm^2

+Average of the presented benchmarks

includes a full set of peripherals: Quad Serial Peripheral Interface (SPI), Universal Asynchronous Receiver-Transmitter (UART), GPIO, Joint Test Action Group (JTAG), and a Double Data Rate (DDR) HyperBus interface to extend the size of the on-chip memory. The floor-plan area for Quentin is 2.31 mm^2 and its effective area is 1.22 mm^2.The specifications of this SoC tape-out are shown in Table 1.2 [14].

Table 1.2: Quentin SoC tape-out specifications.

Quentin SoC	
Bits	32
RISC-V user spec	IMFC
Technology	GF 22nm
Area	1.22 mm^2
Target frequency	32 Khz - 938 MHz
Power	300 μW - 66.2 mW
Best performance	2400 MOPS

*MOPS performance are normalized to RV32IMC equivalent operations

Ariane core

Ariane is open-source RISC-V application class in order core. [2] This microprocessor has the support for memory virtualization, privilege modes, and all the necessary RISC-V privilege specification extension to boot a Linux kernel. This microprocessor had been designed to be used in academia. Besides, it has a very liberal licensee permit that Ariane can be used in the industry. The project provides support for Verilator and QuestaSim RTL simulators as well an FPGA implementation and a pre-built Linux image. The authors claim that its tape-out implementation in the GlobalFoundries 22FDX technology node achieves up to 40-Gops/W energy efficiency, which is superior to similar cores presented in state of the art. The main specifications of Ariane Core Tape-out are presented in Table 1.3 [2].

Table 1.3: Ariane tape-out specifications.

Ariane core	
Bits	64
RISC-V user spec	IMC
RISC-V priv. spec	1.11
Technology	GF 22nm
Area	0.3 mm^2
Target frequency	1.7 Ghz
Power	52 mW
IPC	0.87

1.5 Thesis outline

The rest of the thesis is structured as follows. Chapter 2 presents a technical and theoretical background about the different steps in the ASIC development. Chapter 3 describes the design considerations and additions to the DRAC SoC that were performed. Chapter 3 presents the descriptions of the necessary steps to generate the final layout, including RTL implementation, verification, synthesis, physical design, and GLS. Chapter 5 reports the results obtained in the different stages of the work-flow, including the FPGA prototyping. Finally, chapter 6 summarizes the work achievements, gives the obtained conclusions, and proposes some future work to this project.

[2] https://github.com/pulp-platform/ariane

Chapter 2

Theoretical Background

This chapter gives, technical and theoretical background of the main stages of the design and implementation of an ASIC tape-out. The basic theoretical concepts in the different stages are described, including RTL design, logic synthesis, physical design, RTL verification, post-synthesis verification, and DFT.

2.1 ASIC design process

Before the fabrication of a IC, several stages of design must be covered. These stages are listed in Figure 2.1, these stages implied different levels of abstractions. In the design abstraction hierarchy, a higher-level description has fewer implementation details but more explicit functional information than a lower-level description, it means that at higher levels is easier to understand and analyze the functionality of a system while lower levels, metrics, and simulations are closer to the actual behavior of the IC. Typically, Hardware Description Languages (HDLs) are used to cover most of these stages. Modern HDLs, such as Verilog and VHDL, have been developed to cover several levels of the abstractions from the system-level design to almost all the physical level design [15].

The design flow for a IC, is as a series of steps through the different levels in the design hierarchy. These steps are listed below.

- Specification. It includes the definition of system specifications and requirements. It can include some Electronic System Level (ESL) design.
- Behavioral simulation. This step verifies the ESL design and validates an implemented model. Also, it can give metrics to feedback on the system specifications. ESL design is commonly described in C, C++,

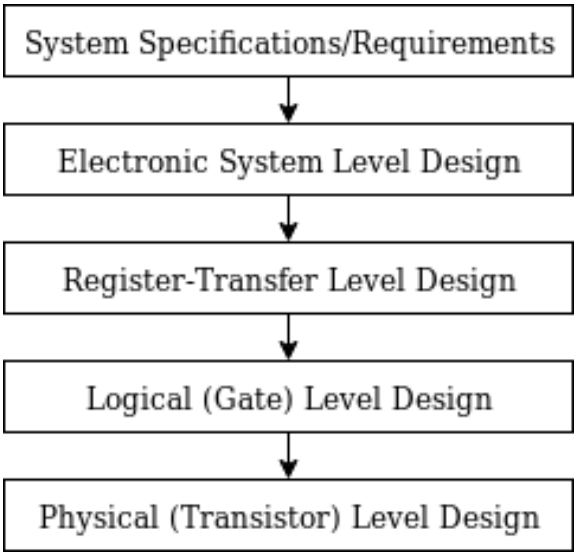

Figure 2.1: Design hierarchy stages.

SystemC, SystemVerilog, or a mixture of theses languages, modern verification and simulation tools can either convert the language to VHDL or Verilog or directly accept the language constructs.

- RTL coding and modeling: This step is the implementation of a system that is detailed enough to be synthesized. The use of multiple languages to describe this level of design in the same project is a common practice.

- Logic synthesis. This step is a transformation that conduces to lower levels in the design hierarchy, including gate-level design, transistor-level design, and physical design. Logic synthesis is a generic process that can be highly automated by the use of EDA tools. It also is used to obtain physical metrics, such as area, max frequency operation, and power consumption estimations. These metrics are used to feedback the ESL design and the RTL level design.

- Verification. This step includes the design and modeling of the necessary functional tests that provide the desired coverage for an electronic system. It also can be considered as the most critical aspect of the IC development, because it requires more time and effort than other stages. The implemented functional tests must be applied to the different design levels and must be passed by all of them.

- Physical Design. This stage includes the floor-planing process, PnR, and the Mask Generation output in GDSII format.

- Device Fabrication. Fabrication is an industrial step that is performed by different foundries. Typically several chips of the same technology node are fabricated at the time, and one foundry is specialized on few fabrication technology nodes.

- Post Silicon Validation. In this stage, testing is performed to detect failures that can occur in the fabrication process. These failures can be exposed immediately or after some period of working time. Several techniques must be considered at higher level designs to provide some grade of controllability and observability once a IC was fabricated. This techniques include DFT, Built-In Self-Test (BIST), fault simulation, Automatic Test Pattern Generation (ATPG) and others. Modern EDA tools provide integral solutions to implement DFT features to an electronic system.

These steps and their interactions are shown in Figure 2.2 [15].

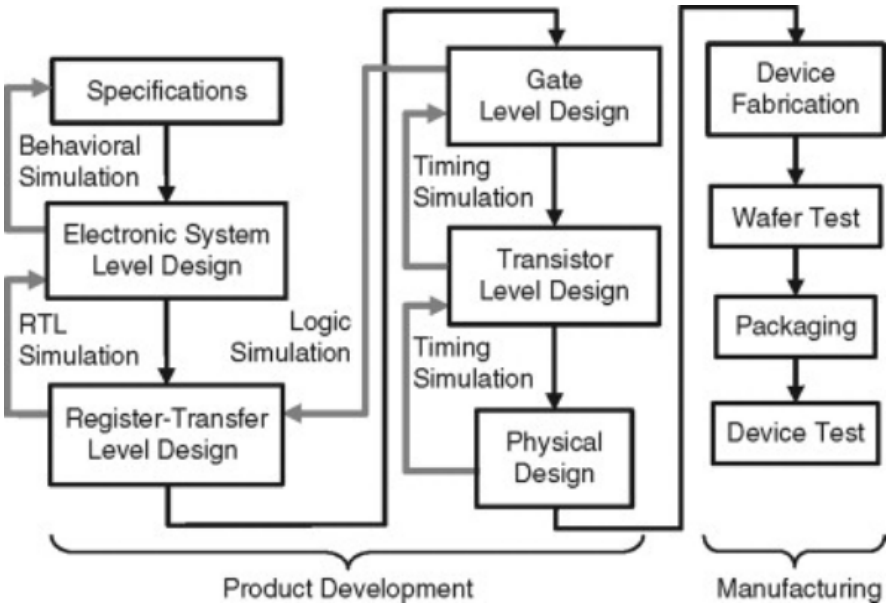

Figure 2.2: IC design, verification and fabrication flow.

2.2 Electronic design automation

EDA exist because of the necessity of shrink the rapidly growing "designer productivity gap" that exists between how many transistors we can manufacture per chip, and how many person-years we need to complete a design

with that many transistors [15]. Moore's Law has implied a constant growth in the complexity of IC design; this constant growth has enforced the necessity of EDA. The EDA market for ICs had grown to approximately $3B with Cadence and Synopsys as the main commercial players [16]. This growing market has developed more and more sophisticated EDA techniques and tools over time. In the following sections, a brief description of the EDA tools and methods is given.

EDA tools can be applied in different areas of electronic system design; some of them are listed below:

- Schematic entry and documentation.
- Printed circuit board design, interactively or automatically.
- FPGA and Programmable Logic Device (PLD) design.
- Logic design and synthesis.
- Circuit simulation, digitally and analog.
- ASIC design, only digital design.
- ASIC design, mixed-signal with analog library cells.
- Design of full-custom cells.
- Integration of semiconductor Intellectual Propertys (IPs), SoC design.

EDA is an area in which computers do a substantial part of the engineering process. However, EDA cannot replace part of the experience of the engineer, but they can be used to increase an engineer's productivity. Besides, Semiconductor technology and EDA are tightly interlocked and depend on one another, progress in EDA leads to progress in semiconductor technology and vice versa [17].

EDA provides to chip designers a design method, which can be considered as a set of complementary design tools built on a design abstraction, as well as a set of processes and guidelines that indicate the flow of the design. The SW design tools include design entry tools, which capture design specification, design synthesis tools, which target different parts of the design specification and bring them down to low-level implementation; and verification tools, which either simulate/verify the specification or compare a specification against its implementation. EDA brings the possibility of handle more complex systems by incrementing the level of abstraction at a

system is designed and modeled. In this way, designers interact with fewer but more complex components of a system.

As a result of current EDA capabilities, the design process is highly automated from the point where the models reach a level of detail that can be processed by logic synthesis. This automation is particularly true for physical design but to a lesser extent for manufacture test and test development. EDA synthesis options include features and capabilities to select and control area and performance optimization for the final design. Design considerations include designer experience and EDA SW availability and capabilities.

EDA algorithms, techniques, and SW can be portioned into three distinct categories:

- Logic design automation
- Testing
- Physical design automation

Logic design automation refers to all modeling, synthesis, and verification steps that model a design specification of an electronic system at ESL, verify the ESL design, and then compile or translate the ESL representation of the design into an RTL or gate-level representation.

Testing was created due to the necessity of maintaining high levels in the reliability of the more and more complex systems and at the same time to reach proper levels in the efficiency of the testing process. As a result, it has become a requirement that DFT features be incorporated in the RTL or gate-level design before physical design to ensure the quality of the fabricated devices.

Finally, physical design refers to all synthesis steps that convert a circuit representation into a geometric representation (GDSII). Modern physical design typically is divided into three major steps [15].

- Floor-planning
- Placement
- Routing

In further sections, we will review what are the main stages in IC design and how EDA tools are applied to these stages. These sections include Modeling and Verification that are covered by logic design automation tools, physical design automation tools, and testing tools (DFT).

2.3 Modeling

Modeling uses the design specifications as a starting point. It uses them to develop a behavioral description of a system using ESL languages, such as SystemC, SystemVerilog, VHDL, Verilog, and C/C++, then the description is simulated to determine whether it meets the system requirements and specifications. The objective of modeling is to describe the behavior of the intended system in several behavioral models that can be simulated for design verification and then translated to RTL for logic synthesis. During design verification, several iterations of modeling and simulation steps are usually required to obtain a working behavioral description for the intended system to be implemented [15]. Modeling, as part of the EDA design tools, was created to describe a complex IC at high levels of abstraction. The most popular choices of languages for digital systems development are SystemVerilog, Verilog, and VHDL. These languages have been updated over the years to add more features and cover more levels of abstraction than the original versions were intended. Nowadays, Verilog is used to represent the netlists in the lower layers of the physical design, and SystemVerilog, as an extension of Verilog, can use most of the object-oriented constructs to describe a system at ESL.

We can divide Modeling languages into two main categories. The first category is conformed by the ESL modeling languages, which include the languages that describe a system at ESL and are synthesized using High-Level Synthesis [18]. The other category is the RTL modeling languages that are processed by the logic synthesis process.

2.3.1 ESL languages

In the ESL languages, there are two kind of methodologies that can be used at ESL. The function-based ESL methodology and the Architecture-based ESL methodology.

Function-based ESL design

The function-based ESL design method uses a computational model to compose different functional components into a complete system. The computational model determines how the components execute in parallel and how they communicate with each other. One of the most widely used ESL computation models in the industry is Simulink developed by Math Works. Simulink graphically captures a system as a connected network of components, also called a block diagram. Here each block captures the instan-

taneous behavior of a component, i.e., how the output of the component changes given an input and a state variable.

Architecture-based ESL design

The second methodology in ESL languages is the architecture-based ESL methodology. This methodology follows the traditional discipline of computer organization closely. Here design is conceptualized as a set of components. Because these components are often available as reused designs, either from previous projects or acquired from third parties, they are referred to as intellectual property or IPs. The components are often connected through buses, switch fabric, or point-to-point connections. Communication in the architecture-based method is abstracted as a set of transactions [15]. One language that can be used with this methodology in the Bluespec language, Bluespec provides an approach to high-level synthesis that is widely applicable across the spectrum of data and control-oriented blocks. Bluespec is explicitly parallel and based on atomic transactions. Atomic transactions encompass communication protocols across module boundaries, enabling robust scaling to large systems and robust IP reuse [19].

2.3.2 RTL languages

The other kind of languages are RTL modeling languages. These languages required a more detailed description by the designer. In general, they are more limited to perform high abstraction features that are common in modern software languages. However, as is mentioned in section 2.1, RTL languages have been widely adopted by the industry to cover most of the IC design process, including RTL design, gate-level design, and transistor-level design. Verilog and VHDL are Institute of Electrical and Electronics Engineers (IEEE) standards [20, 21]. The standardization of these languages permits that the same code can be used by the majority of the industrial SW design tools. Originally Verilog and VHDL have been developed to simulate circuits and later have been adapted as the languages to describe a system for the logic synthesis process. Because of this, only a subset of these languages is capable of being synthesized by a logic synthesis tool. Also, the support of some constructs can differ depending on the target technology and the synthesis tool, e.g., FPGA or ASIC. In contrast, SystemVerilog was created as an extension of Verilog. SystemVerilog is an unified hardware design, specification, and verification language. The SystemVerilog standard includes support for behavioral, RTL, and gate-level hardware descriptions. Also, SystemVerilog is provided with constructs for verification such as testbench,

coverage, assertion, object-oriented, and constrained random constructs. Finally, SystemVerilog provides Application Programming Interfaces (APIs) to foreign programming languages; in this way, SystemVerilog can be used with models written in languages that are closer to the ESL design.

2.4 Verification

Design verification is the most important aspect of the product development process, consuming as much as 80% of the total product development time. Because less effort is required to obtain models that can be simulated but not synthesized, design verification can begin earlier in the design process, which allows more time for considering optimal solutions to problems found in the design or system. [15]

Functional Verification is a comparative process. It includes a wide set of techniques to find faults in the behavior of a device. Functional Verification shows if the hardware or software meets the original specification requirements. Finally, the functional verification dos not show the fault itself. It merely shows the presence of an error [22].

It is easy to confuse the concepts of testing and verification, even in some cases they are used as synonyms. It is important to remark the difference between these two concepts and how they are applied in the IC design process. Verification is a comparative process with a model to find mismatches with the functional specifications. Functional Verification assumes a correct functionality in the interconnections and tries to detect errors in the logic's implementation and design. On the other hand, testing is a series of procedures that apply a stimulus to a Device Under Test (DUT) based on the design specification and fault models that are applied to find failures in the behavior or the performance according to the expected design. Most fault models in testing try to model failures in the interconnections of a circuit element, assuming the correctness in the implemented logic. Designers of functional Verification are closer to the highest levels of abstraction more closer to the system specifications and requirements; Testing designers are closer to the ESL design, RTL design, and gate-level design. A simplified diagram of the circuit design flow and how verification and testing intact in this flow is shown in Figure 2.3.

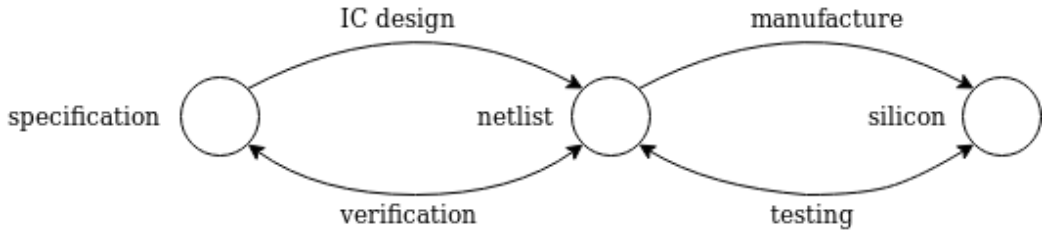

Figure 2.3: Verification and testing as part of the IC design flow.

RTL Simulation

Gate Level Simulation

2.5 Design for Testability

Testing is a set of tests applied to the DUT to determine if the behavior in each test meets the specification. i.e., testing consists of a set of stimuli applied to the device to test a particular test case or analyze the response of the performance based on the expected behavior [22].

Test development consists of selecting specific test patterns based on circuit structural information and set of fault models [15].

2.6 Logic synthesis

Designers must be careful to avoid constructs in HDLs that allow the model to self initialize but can not be reproduced in the final circuit by the synthesis system. This description has special importance in control signals that are not property initialized and would result in an uncertain behavior in post-synthesis simulations and random behaviors in the IC. Good coding and reusability styles, as well as user-defined coding style rules, play an important role in avoiding many of the synthesis errors. [15]

Chapter 3

System on Chip Design

This chapter will review the details and considerations about the design and implementation of the DRAC SoC. It will be covered the drawbacks and constraints that implied modifications to the original FPGA implementation of the project.

The DRAC SoC was a modification of the *lowRISC* project in order to put the Lagarto I Core architecture. The project had been adapted in order to fabricate it in a 65nm. The following sections of this chapter present the design and implementations that were developed for the *preDRAC* project.

The *preDRAC* project implied a joint effort from different teams by people from the different institutions involved. In this chapter, we will review the different parts of the design, and the present document focuses on those parts related to the objectives of the thesis. In Section 3.1 an overview of the final complete design is described. In Section 3.2, all the design considerations that must be taken into account are listed and explained.

The design and implementation of the *preDRAC* SoC comes from different sources, these sources are indicate in Table 3.1. The modules denoted as internal sources where developments and implementations performed by the DRAC development team.

3.1 DRAC SoC overview

Figure 3.1 shows the block diagram of the *preDRAC* single-core processor. This design incorporates a 5-stage single-issue in-order Lagarto pipeline that implements the 64-bit RV64IMA scalar RISC-V ISA, as well as the associated instruction and data caches, a unified L2 cache and the peripherals required to connect the processor with external devices such as main memory, JTAG, UART and an Secure Data (SD) card, as described in detail in the next

sections.

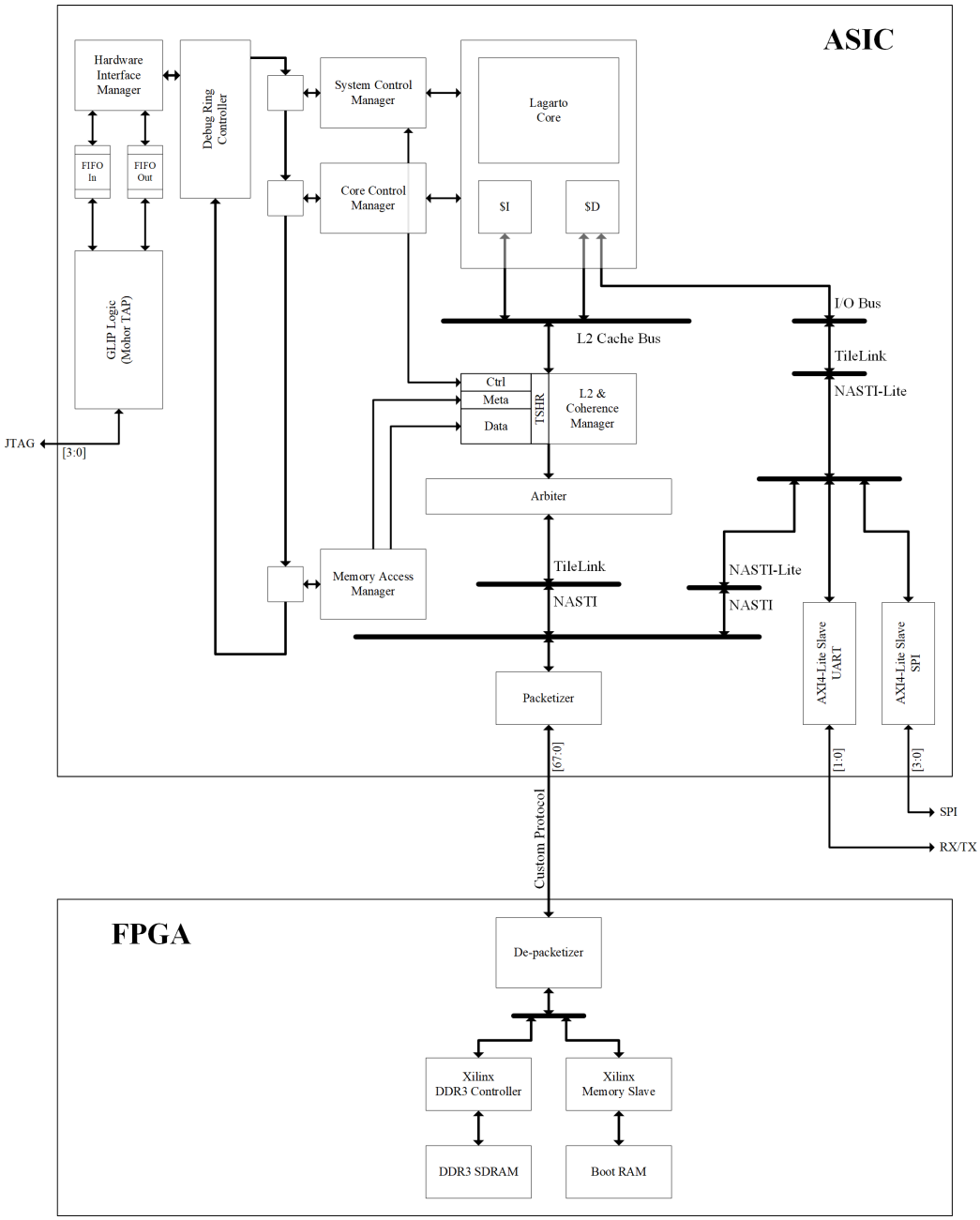

Figure 3.1: Drac SoC block diagram.

Finally, Table 3.1 summarizes the main IP blocks in the preDRAC processor design together with a short description of the block, the source of the code (including parts of the design that are based on external projects), and

language of the block.

Table 3.1: *preDRAC* processor IP blocks

IP Block	**Description**	**Source**	**Language**
Lagarto core	RV64IMA 5-stage in-order pipeline, Bimodal Branch Preditor with 1024 entries.	internal	Verilog and Chisel (CSRs)
Instr. cache	4-way 16KB, 2-cycle access latency, VIPT, 64B cache blocks, 8-entry TLB.	open, lowRISC 0.2	Chisel
Data cache	4-way 16KB, 3-cycle blocking access latency, VIPT, 64B cache blocks, 8-entry TLB.	open, lowRISC 0.2	Chisel
L2 cache	8-way 64KB, 3-cycle access latency, PIPT, 64B cache blocks, MESI protocol.	open, lowRISC 0.2	Chisel
TileLink	128-bit wide 0.3.3 version.	open, lowRISC 0.2 [7]	Chisel
UART	AXI4-Lite Slave interface, 11 bit per packet, configurable baudrate, parity bit and stop bits, up to 3MBauds.	based on Lagarto SoC v1.0, internal	Verilog
SD Card controller	AXI4-Lite Slave interface, Bidirectional 8-bit wide SPI miso/mosi packets, up to 25Mbps.	based on Lagarto SoC v1.0, internal.	Verilog
JTAG	Communication interface between (PC) C read/write functions and (Internal) in/out FIFOs, uses an FT2232H transciever, clock at 50MHz.	open, GLIP, OpenOCD and MohorTAP.	Verilog, C
Packetizer	AXI-4 front end interface, 64-bit wide, 50 MHz in 65nm TSMC standar I/O	internal.	Verilog
PMU	9 counters, user accessible.	internal	Verilog
Debug ring	start/stop execution, read/write register values, write program to L2 cache.	internal	Verilog

3.2 Design Considerations

3.2.1 Physical level considerations

Originally the DRAC SoC was tested and prototyped using the KC705 FPGA board. To deploy the project into an ASIC, it was necessary to take into account the physical constraints of the 65nm target technology. These restrictions include area, number of IO pins, and physical interfaces.

Area

The main restriction was the area of the chip. The design was restricted to the size of one section if the design oversize that restriction it had been implied more cost to the IC fabrication.

Number of IO pins

The area of a IC is related to the maximum number of pins that are possible to use depending on the chip's perimeter. With the dimension that the Drac tape-out has, It has the restriction to around 100 IO pins.

Physical interfaces

Finally, the last physical restriction regarding the target technology was the physical connections that it is possible to use. High bandwidth peripherals such as DDR main memories, can not use regular IO connections. They require special tuned analog circuitry for a given fabrication technology node, and a given range of frequencies, e.g., an IP of a physical DDR3 interface designed for a 28nm technology, can not be reused in a 65nm technology. This strong attachment to the fabrication technology also applies to high-speed differential communication protocols [23, 24].

3.2.2 RTL considerations

As is said in Section 1.1, an ASIC fabrication implied more considerations in the design and the implementation of a project. At RTL design, it was necessary to perform some changes in the code in order to fit these restrictions, besides removing not synthesizable constructs.

Reset Strategy

The first design decision that was taken in order to have good results in the post-synthesis verification was to decide a homogeneous way to reset the

following elements of the system [25]. Having a mixed implementation of the reset control in the internal flip-flops leads to strange constructs and unsuspected behavior in the pots-synthesis verification phase, these causes difficulties to debug. Also, all the control signals that are saved in registers must be properly initialized to a known value with the reset signal. Finally, all reset signals must be synchronized to the main clock, a not correctly synchronized signal lead to metastability problems. Synchronous reset signals must be synchronized with sampling registers connected to the same clock source of the logic. Asynchronous reset signals can be activated at any time but should be released at the edge of the clock signal. A proper synchronizer circuit must be implemented for this purpose; Figure 3.12 shown an example of an asynchronous reset synchronizer.

Clock domain crossing

Another important consideration in the RTL design and implementation are the clock signals connected to the registers. All the different clock domains used in the system must be identified, and unnecessary clock divisions must be avoided, they can be replaced by synchronous counters and enable control signals. When a clock crossing implementation is necessary, it must be implemented with a Clock Domain Crossing (CDC) logic such as sampling registers or dual-clock asynchronous First In, First Outs (FIFOs) in order to avoid metastability problems. Modern EDA tools provide CDC analysis to find potential errors in the design.

3.3 SRAM Macro-cells

In order to fit the area limitations it was necessary to use SRAM macro-cells. First all the ram structures that can be replaced by SRAMs were identified, these are shown in Table 3.2.

All the used SRAM were one read port and one write port. In the case of the Lagarto I core, the structures that were decided to be implemented as SRAMs are limited to the Bimodal branch predictor tables and the dual-port register file. The rest of the SRAMs in the SoC were the cache memory banks and the Translation Look-aside Buffer (TLB) memory tags.

SRAMs are usually used to implement large memory structures such as Level 2 Cache (L2) and L3 memories. The main reason for this is because SRAMs usually becomes at a high cost in latency. It is common to implement different clock domains for large multi-ported SRAM based memory structures [26]. In the case of *preDRAC* SoC we decided to use only one

Table 3.2: Sram macro-cells used in *preDRAC* SoC.

Structure	**Depth**	**Width**	**Size** KiB	**Area** mm^2	**Instances**
L2	4096	128	64	0.67	1
L2 TLB tags	128	176	1.38	0.10	1
L1-D banks	256	128	4	0.09	4
L1-D TLB tags	64	88	0.69	0.04	1
L1-I banks	256	128	4	0.09	4
L1-I TLB tags	64	80	0.63	0.04	1
Integer RegFile	32	64	0.25	0.01	2
BIPC	1024	28	3.50	0.05	2
BTB	1024	40	5	0.07	2
PHT	1024	2	0.25	0.01	2
Total core			18	0.29	
Total SoC			116.69	1.84	

frequency domain and use SRAM macros as much as possible to optimize the area of the IC.

3.4 Core Pipeline

Lagarto I is a 64-bit in-order single-issue scalar core based on RISC-V ISA. The design is composed of five pipeline stages: fetch, decode, read registers, execution/memory-access, and write-back. Figure 3.2 shows a block diagram of the Lagarto I microarchitecture. The main features of Lagarto I are:

- In-order five stages pipeline.
- 64-bit RV64IMA RISC-V ISA.
- RISC-V Privileged Spec-v1.7.
- Implements a Bimodal Branch Predictor (1024 entries).
- Register File with 2 read-port and 1 write-port (64-bit words x 32-entries).
- Precise Exception scheme.

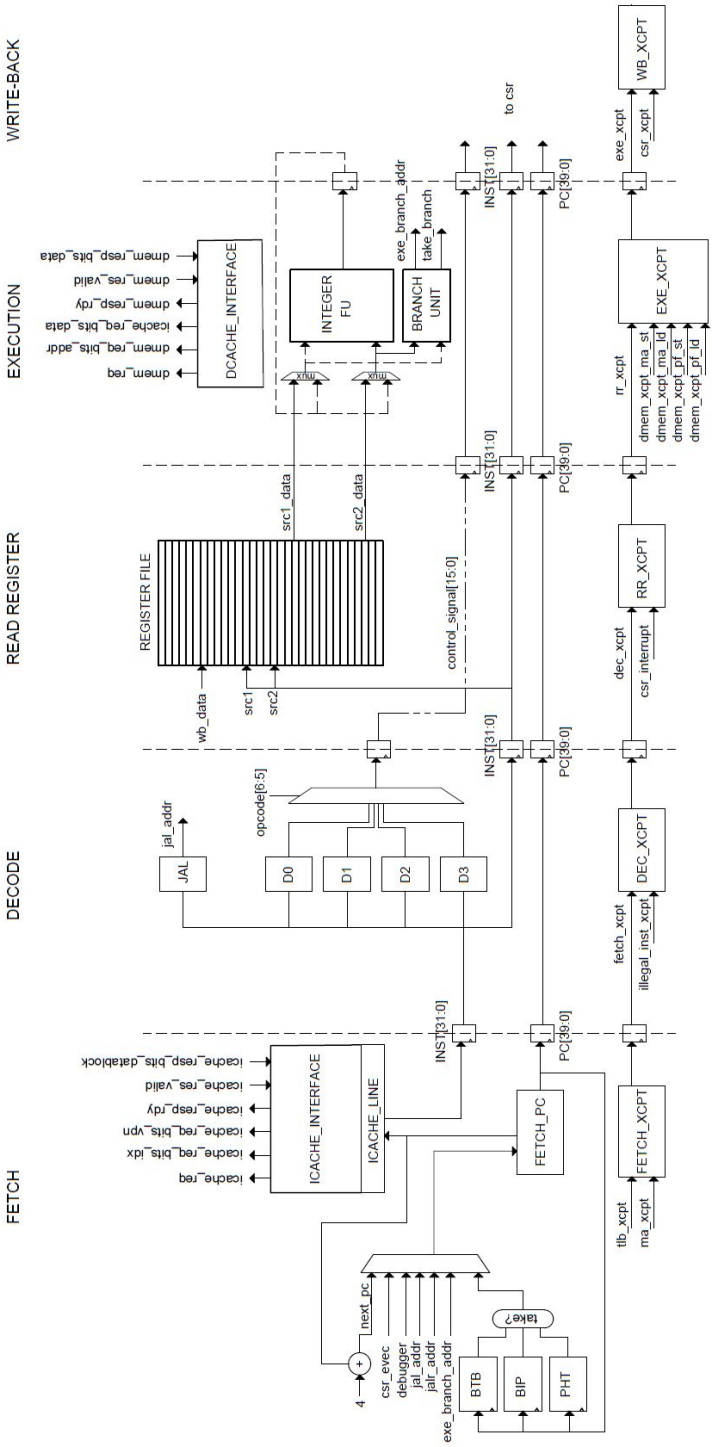

Figure 3.2: Lagarto I Microarchitecture

This in-order core was implemented and designed by Cristobal Ramirez Lazo, based on previous developments by the CIC-IPN.

3.5 FPGA-ASIC Design Split

Ideally, the objective of the tape out would be to include the whole *preDRAC* SoC, together with all the peripherals, including the memory controllers on a single die. However, we faced many challenges regarding the availability of certain technologies and IP for fabrication. Due to the lack of both a Phase-Locked Loop (PLL) IP to generate a High-speed clock or a differential connection physical interface to connect an external high-speed clock generator. The maximum clock speed that we can use is restricted by external oscillators that are interconnected through a single-ended pin connection. Because of these restrictions, the target frequency of the chip was defined to be 200MHz.

Design Motivation

On the FPGA prototype, the Xilinx IP that we used were: DDR3 controller, SPI, UART, and of course, a PLL for clock generation. In the case of the DDR Random-Access Memory (RAM), although it was possible to find a DDR controller for our SoC, we lacked access to a physical layer implementation that would be required to be able to include the DDR3 Synchronous Dynamic Random-Access Memory (SDRAM) on our chip to be taped out. Additionally, the SPI and UART IPs from Xilinx were replaced by IP counterparts that were designed by the DRAC team. These constraints required our SoC design to be partitioned in two: an ASIC part and an FPGA part. Since we had to have the DDR3 RAM on the FPGA, we needed a certain bus connection between the ASIC and the FPGA. We also lacked Serializer/Deserializer (SerDes) transceivers to implement high-speed buses, so we opted for a simple FMC single-ended connection option. Although this is a low throughput connection, we decided that it was sufficient for a proof-of-concept implementation.

Through this FMC connector, we implemented a *packetizer* that breaks down AXI transactions into packets and sends them through a narrower link. Later, once the FMC link is traversed, a depacketizer reconstructs these packets back into AXI transactions.

Another issue that we faced was clock signal generation since we did not dispose of a PLL to include in our ASIC design. One option was to use the PLLs available on the FPGA board to generate the clock signal and to

send it to the ASIC. According to our tests, FMC's operating frequency was between 25Mhz and 50 Mhz, depending on the length and the quality of the cable. We discarded this option. Instead, we chose to generate a 200 MHz clock using an external oscillator and to clock the ASIC.

3.5.1 AXI implementation

The PLB of the *lowRISC* SoC is an implementation of the *AXI* protocol [27]. That implementation is called *NASTI*, is open source and, is licensed by the *University of Cambridge*. The diagram of the PLB *NASTI* implementation of *lowRISC* is shown in Figure 3.3 [7].

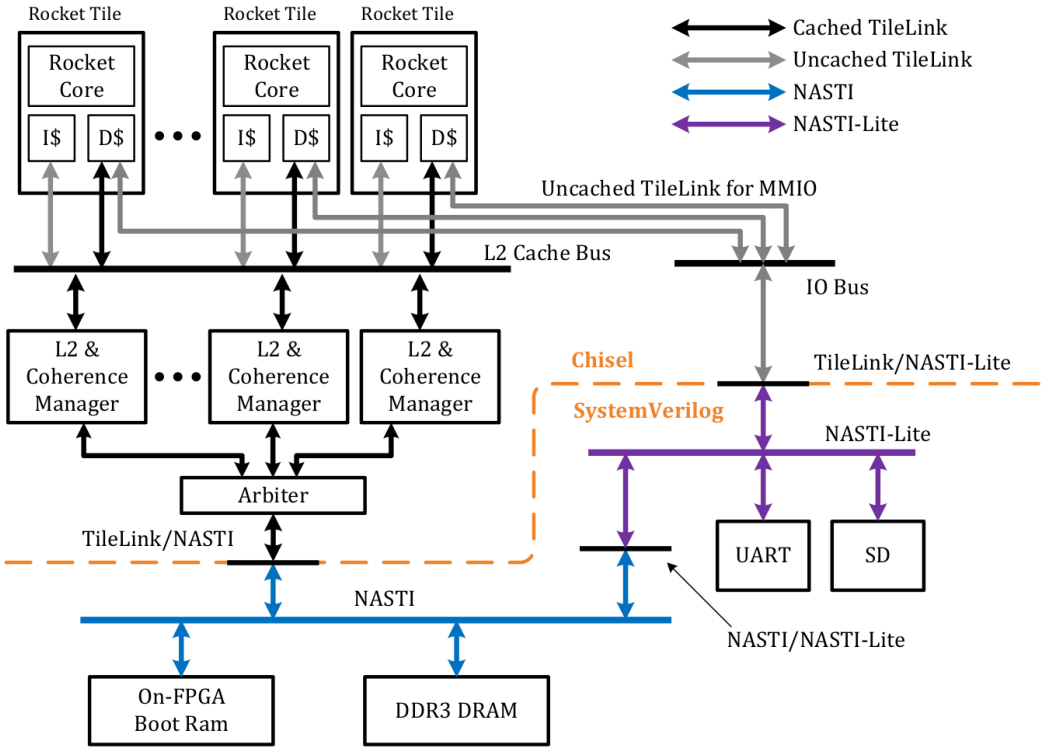

Figure 3.3: *lowRISC* SoC.

Due to the lack of a main memory physical interface, it was proposed to use an auxiliary FPGA board to use its embedded physical connection to a *DDR3* memory module. In this way, the system architecture was split into two parts. The main memory controllers were located in the FPGA board and, the rest of the design was in the custom fabricated ASIC. The *NASTI* implementation of the *lowRISC* SoC was used as baseline to build the decoupled PLB of the DRAC SoC as is shown if Figure 3.4.

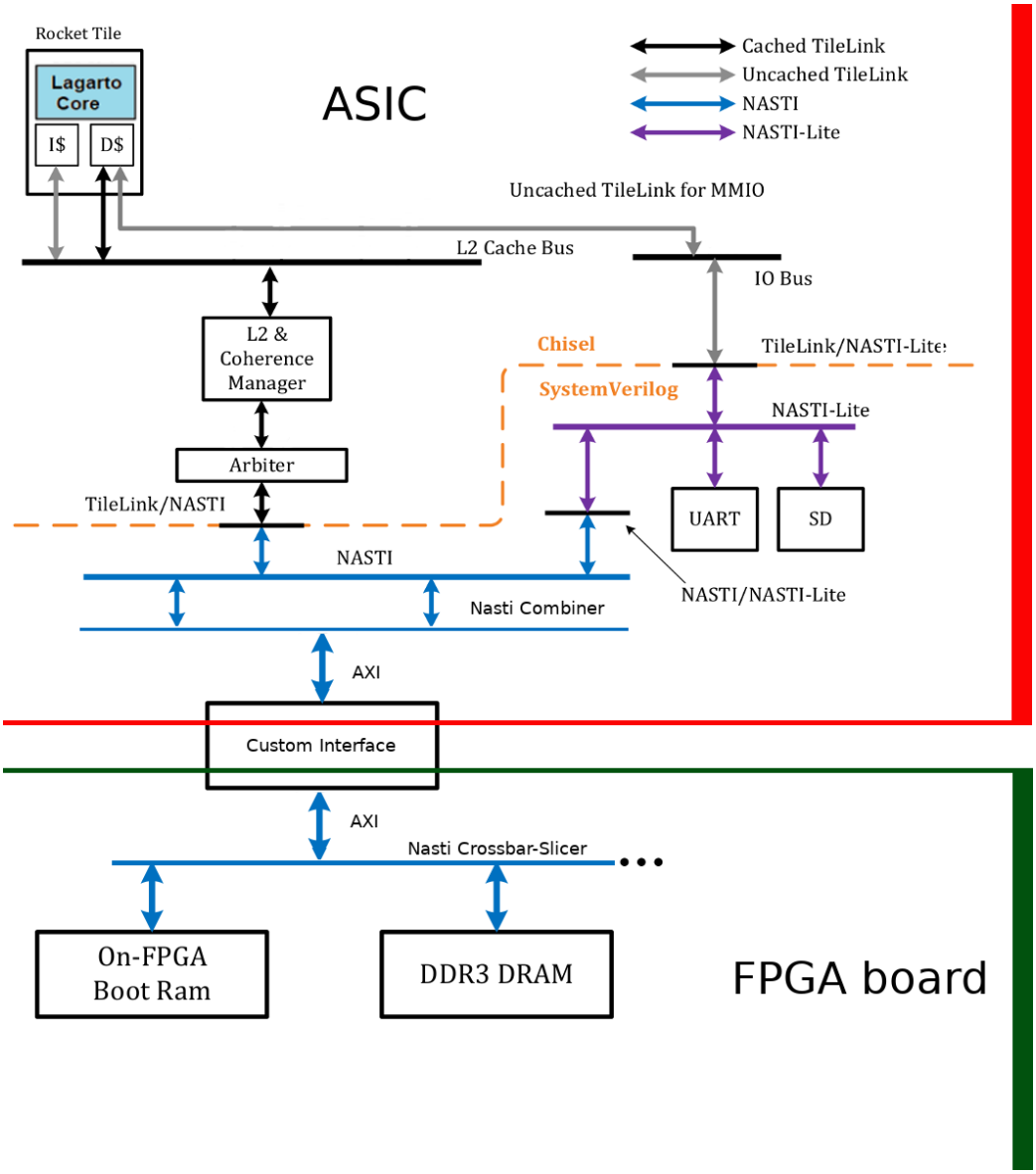

Figure 3.4: Simplified block diagram of the modified system to communicate it to a external FPGA board.

From a general point of view, there were two main changes to the original *lowRISC* implementation. First, the new design includes a *Combiner* logic for the two main memory channels (the *boot ROM channel* and the *DDR3 DRAM channel*); the output of that *combiner* was connected to a custom interface that interconnects the both parts of the design, *FPGA side* and *ASIC side*; this combiner is shown in Figure 3.4 as *NASTI Combiner*. Sec-

ond, was that a second level of *Crossbar-Slicer* logic were added in the *FPGA side* were the memory controllers are connected; this is shown in Figure 3.4 as *NASTI Crossbar-Slicer*.

The *Combiner* logic takes two or more *AXI* interfaces and converts them into a single group of signals. The *Slicer* logic performs the opposite function, it generates several *AXI* interfaces from a single group of signals. Finally, the crossbar logic is the module in charge of generating the necessary interconnection among the multiple *AXI* Master and Slave interfaces by using multiplexers, demultiplexers, and buffers. The modifications of the *lowRISC NASTI* implementation were made by Guillem Cabo, part of the DRAC Development Team.

3.5.2 Main memory access

One of the major challenges of porting the FPGA implementation of the SoC into an ASIC was the lack of the physical interface to use a DDR3 memory controller as the system was originally designed. This lack of physical connections within the limited number of input and output pins that the chip could have due to its area limit, around 2.4mm^2, involved the exploration of different solutions. Some of these solutions were the use of different types of memories, such an external SDRAM or an external *Cypress HyperRAM* [28] module. The use of these memories implied less capacity, performance reduction, and the necessity of performing modifications to the Linux kernel in order to fit it in the new memory scheme. Also, a feature of the JTAG debugger interface described in Section 3.6, provides the possibility of insert code directly the L2 avoiding the communication to main memory, was added, but, with this scheme, the Linux boot was not possible. In order to preserve the Linux boot feature, a custom interface was designed and implemented. This interface will supply the functions of pack and serialize the AXI transactions performed by the core and were denoted as *Packetizer*. The main objective of this interface was to use the physical DDR3 memory from an external FPGA board. This feature implied to split the original design into two parts, one that would contain the memory controllers to access to main memory, and a second one that would contain the core and rest of the uncore system, including Level 1 Data cache (L1-D), Level 1 Instruction cache (L1-I), and L2. A simplified diagram of the modified system is shown in Figure 3.4.

Projects such as the *High BandWidth InterFace (HBWIF)* were evaluated to be used in our system. *HBWIF* is an on-chip memory interconnect protocol that is used to communicate to an off-chip FPGA [29]. This interconnect protocol uses an 8b/10b Encoder/Decoder that converts a byte-wide data

stream of random 1s and 0s into a DC balanced stream of 1s and 0s with a maximum run length of 5. A DC balanced data stream proves to be advantageous for fiber optic and electromagnetic wire connections [30]. Although *HBWIF* is an open-source project and the source code was available, it was designed to be used with high-speed serial links, and an analog front end IP was required [31]. Also, the provided digital back end, that packs and serializes the data transactions were based on a different version of *TileLink* [32] than the used by our SoC.

The development of custom analog IP for Very-Large-Scale Integration (VLSI) requires several iterations to perform tests and characterizations on the implemented circuits before a design can be reliable enough to be used in a more complex system. This project was the first try to produce a custom silicon chip; because of these reasons, the decision of using only standard pads and single-ended connections was taken, and it was decided to design and implement a custom protocol that has this restriction.

3.5.3 Test on the FPGA

The Xilinx Kintex KC705 was decided to be used as the auxiliary FPGA board since it was the board where the system was initially implemented and fulfill the required features [33]. The constraints of this interface were that the custom ASIC could not have more than one hundred external IO pins, and the analog circuitry to build differential receivers and transceivers inside the chip was not available. These constraints limit the bandwidth that the communication system could achieve because of the frequency limitations that single-ended connections have.

Before start designing the *Packetizer*, the physical connection between the FPGA board and the ASIC should be specified. We looked at the kc705 specifications [33], the FMC HPC (High Pin Count), was chosen because of its feature of 58 differential user-defined pair connections. Those pairs can be used as 116 single-ended connections, and are specified to use a standard Low Voltage CMOS of 2.5V. Those features fit with the ASIC target technology and the number of pins that it was able to use. To be sure about the FMC connector capabilities, the *Chip2Chip* Xilinx IP application note was consulted. This application note shows how the *Chip2Chip* IP is used as a bridge to connect two AXI-based systems for multi-device system-on-chip solutions. The application note presents a prototype that uses the KC705 board. The Block diagram of the implemented prototype can be observed in Figure 3.5, the implementation makes use of two KC705 FPGA boards and an FMC-to-FMC HPC connector cable as shown in Figure 3.6 [34].

Even though, the *Chip2Chip* Xilinx IP had seemed like the perfect so-

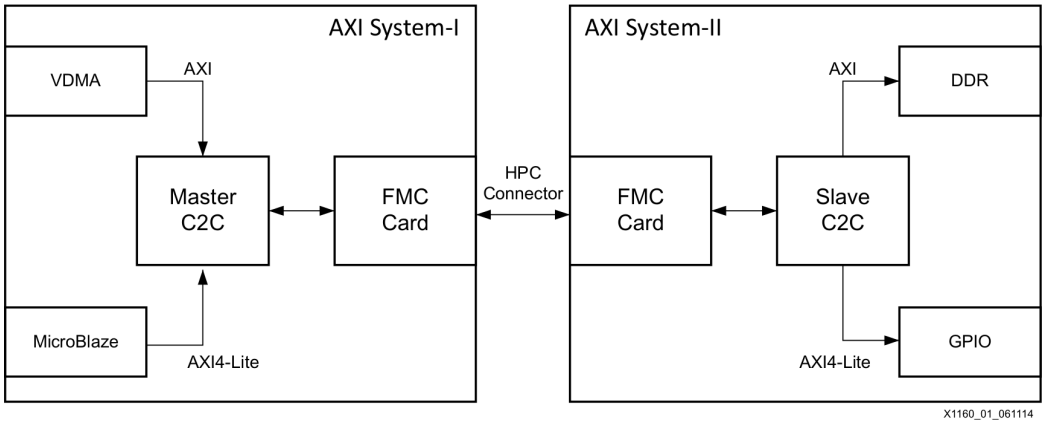

Figure 3.5: Typical AXI Chip2Chip Core Interconnections.

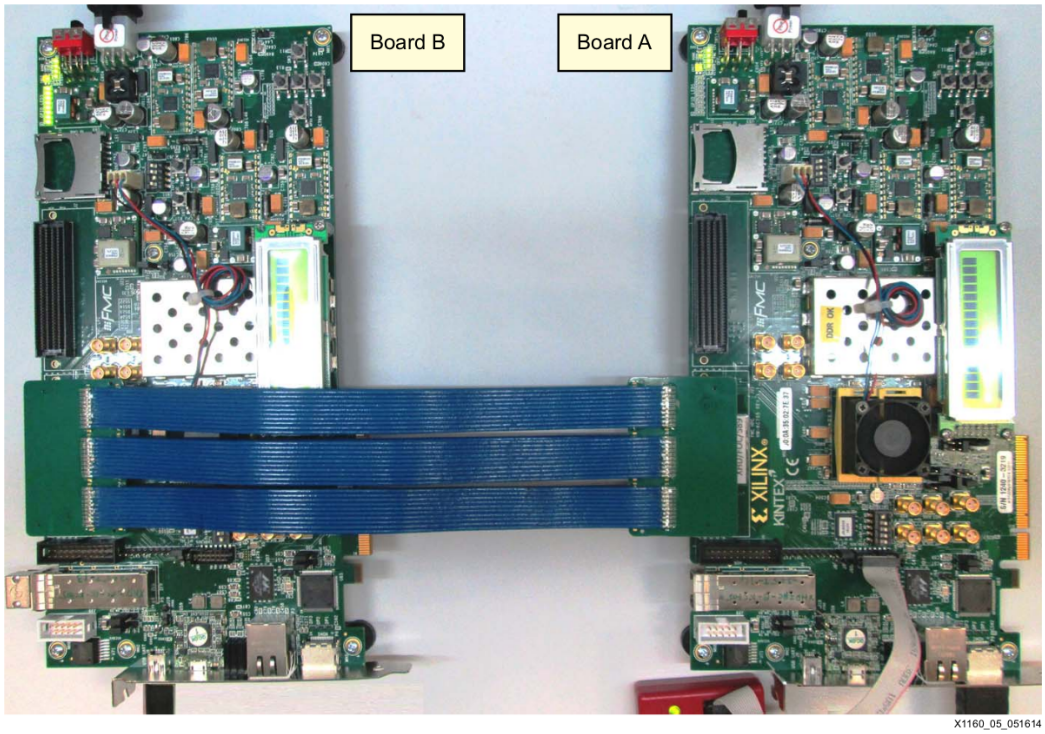

Figure 3.6: Kintex-7 FPGA KC705 Board to Kintex-7 FPGA KC705 Board Setup.

lution to interconnect the fabricated chip with the FPGA to have access to the DDR3 memory, it was hard to port the IP to the specific TSMC 65nm technology, with which the chip was planned to be fabricated, this because Xilinx restricted the access to the source files of the IP. However, it shows

that it is possible to interconnect two independent systems with high bandwidth requirements using the FMC connector and the AXI protocol. We use a pair of KC705 boards and the FMC-to-FMC HPC connector recommended by the application note to test the actual hardware configuration. The test platform consists of two sides one in each FPGA, in one side a counter generates a 16-bit number and sends it through 16 outputs pins connected to the FMC port, then, on the other side, the number is buffered and sent back in the next cycle. The sent number is negated in order to flip the majority of the bits in between transmissions of data. Also, a clock signal was sent from one side to feed the following elements on the other side, with the purpose of test the possibility of sending a clock through the FMC port. To determine if the connection had behaved correctly and was stable, a comparison operation turn on a led when the data received from the other side differs from the sent data. Different experiments were performed at different clock frequencies, using the hardware described before with single-ended connections in the FMC pins. As a conclusion of the experiments was determined that the connection was reliable at a max frequency of 25 MHz.

3.5.4 *Packetizer design*

Once the physical interconnection and its constraints were defined, the control of the communication protocol must be defined and implemented. The *Packetizer* was designed to use as control back-end the handshake specifications of the AXI protocol. The main idea is to hold on channels requests in order to serialize the use of the different channels in the AXI protocol, i.e., use one channel at a time. The AXI protocol uses a Master-Slave scheme. The Master sends *data read* requests and *data write* requests to one or more slaves. The transactions are performed using five independent channels. That are listed below:

- Write address, denoted with prefix AW.
- Write data, denoted with prefix W.
- Write response, denoted with prefix B.
- Read address, denoted with prefix AR.
- Read data, denoted with prefix R.

Address channels carry all the required address and control information for a transaction. The read data channel conveys both the read data and any read response information from the Slave back to the Master. The write data

channel conveys the write data from the Master to the Slave. The write response channel provides a way for the Slave to respond to write transactions. All write transactions use completion signaling All five channels use the same *VALID/READY* handshake protocol. This two-way flow control mechanism enables both the Master and Slave to control the rate at which the data and control information moves. The source generates the *VALID* signal to indicate when the data or control information is available. The destination generates the *READY* signal to indicate that it accepts the data or control information. Transfer occurs only when both the *VALID* and *READY* signals are HIGH [27]. The source must hold the information until the *READY* signal has been asserted by the destination, as is shown in Figure 3.7. The

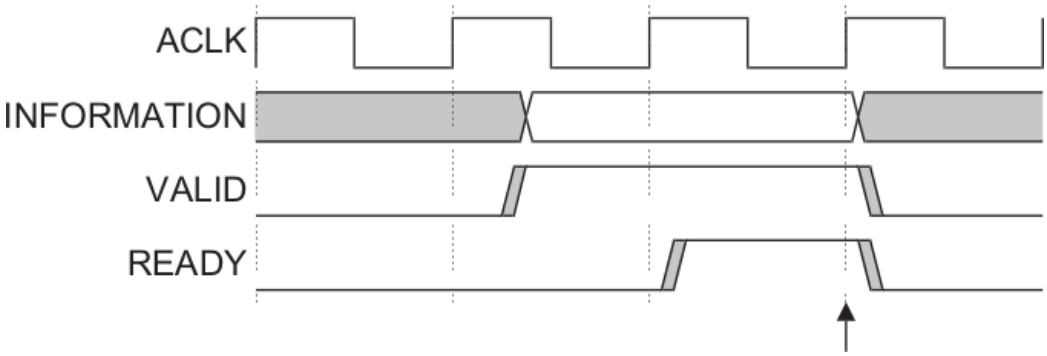

Figure 3.7: *VALID* before *READY* handshake.

AXI protocol also can perform burst transactions to transmit more data with a single address transaction. In this case, several data are sent until the last signal is asserted. Each chunk of data is transmitted as an independent transaction so it can be transmitted with different timing delays. An example of a write burst transmission is shown in 3.8, notice that the response channel is used until all data is transmitted. The main master AXI interface of the SoC has a configuration of 128 bits that correspond to the data width configuration of the DDR3 memory controller in the KC705 FPGA. Also, as is shown in Figure 3.1, the system use two memory controllers that use its own AXI Slave Interface. As mentioned before, the *Packetizer* uses the AXI handshake to pack and serialize the data transfers of each channel. This approach will block the rest of the channels while one request of one channel is being attended. We have chosen to use a bus of 32 connections of the FMC connector pear each direction, i. e. 64 connections in total. Since the transmission rate clock should be reduced because the frequency limits of a single-ended connection and the AXI channels with a configuration of 128 bits of data have a maximum width of 141 bits including the control bits

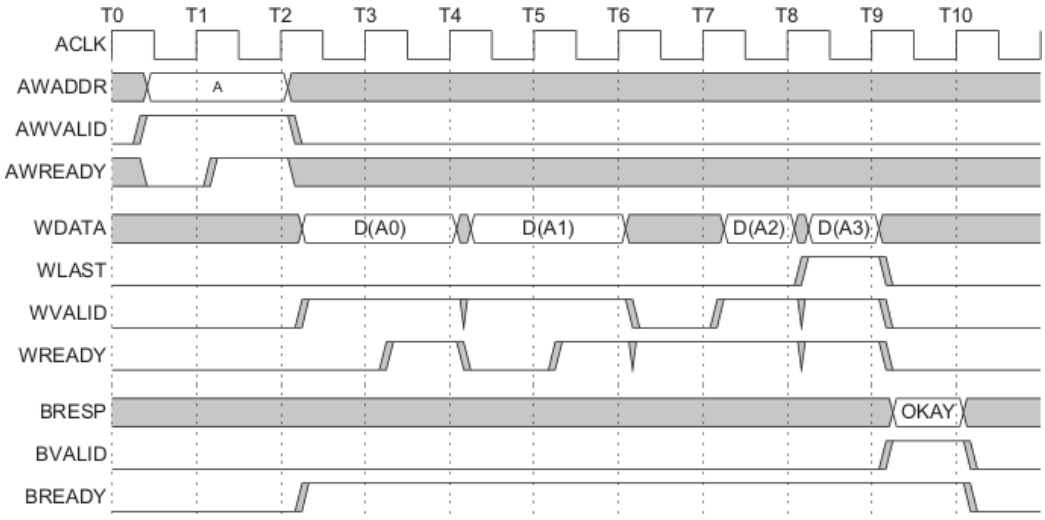

Figure 3.8: Write burst.

of the channel, is evident that a much bigger latency will be introduced to the system in all main memory accesses. This extra latency not only would impact the system's performance; it also can affect the correctness of the system functionality.

To ensure the correctness of this approach, a Verilog implementation that hides the *VALID* signal for a given number cycles and then hides the *READY* signal response for another configurable number of cycles was tested in the complete system. The block diagram of this implementation is shown in Figure 3.9.

This testing scheme implements, in a symmetrical way, three basic blocks, a couple of counters, and a multiplexer. One data-path is for AW, AR, and W channels, these channels transmit data from the Master to the Slave and are in total 427. The other data-path comes from the Slave to the Master and corresponds to B and R channels, and they are 205 bits in total.

The *Valid Buffer* is initialized with zeros, when the *VALID* signal of a specific channel is in zero in the *Valid Buffer*, the multiplexer masks the data bits of the channel and the destination interfaces only receive zeros. The delay count module controls the flow of the *VALID* bits. In the cycle, when the counter reaches its limit, it will allow to pass all the *VALID* signals in the transmitter side to the *Valid Buffer*, the rest of the cycles, the *Delay Count* module halts the *VALID* signals. When the Delay Count module transmits bit to the *Valid Buffer*, it also set the same bits in the *Mask* module. The Mask module hides the *VALID* bits that have already been passed to the *Valid Buffer* to avoid duplicate transactions.

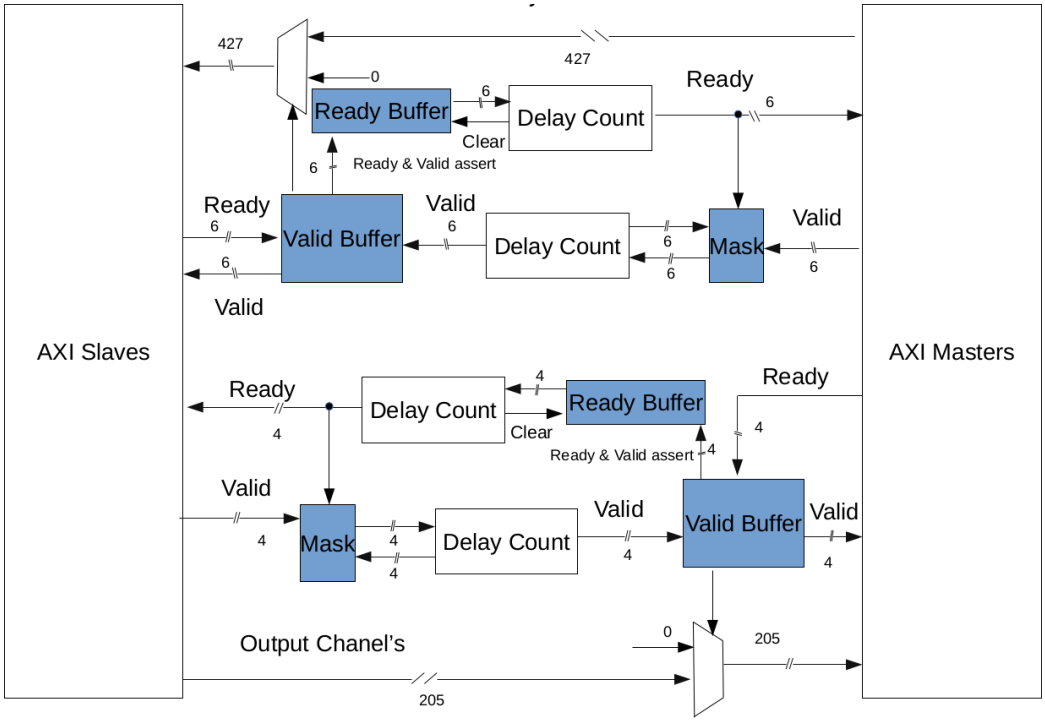

Figure 3.9: *Delay Test* block diagram.

Once a *VALID* bit is in the *Valid Buffer*, the *Multiplexer* allows passing the data bits of the corresponding channel in the same cycle, as is established in the AXI handshake specification.

The bits in the *Valid Buffer* remain in it until the *READY* signal of the destination is received. According to the AXI manual, the transmitter should keep the data in the output port of the channel. When the receiver side of one channel sends a *READY* signal, and the corresponding bit is high in the *Valid Buffer*, the bit in the *Valid Buffer* is cleared and, the *Multiplexer* hides the data of the channel. Finally, the corresponding bit is saved in the *Ready Buffer* module. Another instance of *Delay Count* is used to halt the *VALID/READY* assertion that comes from the *Ready Buffer* to the other side. When the counter reaches its limit, all the bits that are in the *Ready Buffer* become visible to the other side, and the handshake is performed. In the same cycle, the bits that are in the *Ready Buffer* are cleared in the *Mask* and in the *Ready Buffer* itself, in this way the *Mask* will make visible the *VALID* bits of the channels that have just performed the handshake in the next cycle. Allowing another request, since the handshake assertion has been completed one cycle before, if there is another request in the next cycle, it should correspond to a different request.

The *Delay Test* system (Figure 3.9), was inserted in the two AXI master interfaces of the system. In Figure 3.1 these interfaces correspond to *NASTI*, in blue, and *NASTI-Lite* in purple. Once the delay modules were connected between the AXI interfaces, the system was tested with the verification strategy, using the set of ISA tests and the FPGA implementation tests described in Section 4.2. These tests tested the correct functionality of memory accesses, peripherals, and the boot procedure. Then, different delays were inserted by modifying the limit parameter in the Delay Counter modules; several combinations of values were tested with the same test strategy of the ISA tests and the FPGA tests. With these tests was proved that we could add arbitrary delays into the AXI transactions of the system, and it continued behaving correctly.

The next step was to design and implement the control that will pack and serialize the channels of the AXI protocol. Since the communication channel was not able to run at the same frequency of the rest of the system, this control system was connected to some FIFOs that act as buffers for coupling systems that work a different clock domains. The general scheme of the *Packetizer* is shown in Figure 3.10.

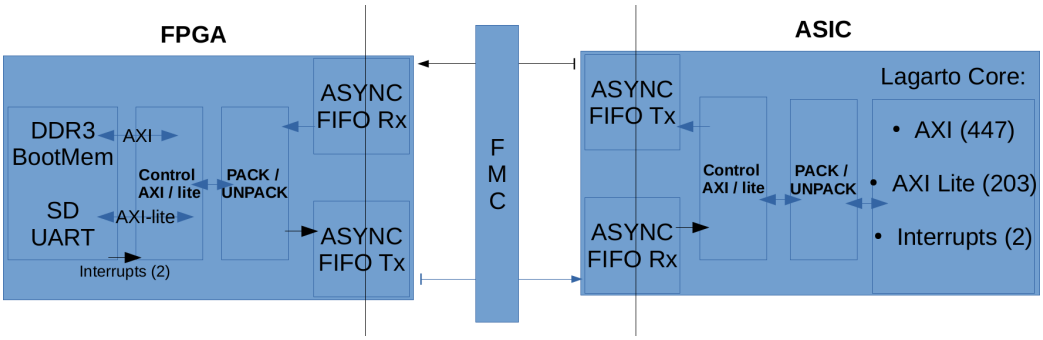

Figure 3.10: Block diagram of the *Packetizer* communication system.

The block diagram of the AXI channels control and the packer and unpacker modules is shown in Figure 3.11. The same *Valid Buffer*, *Ready Buffer*, and *Mask* modules of the *Delay Test* system were used.

As was mentioned before, it was decided to use 32 bits for data output, and 32 bits for data input in the FMC cable, coupled with these data buses a *VALID* and a *READY* signals were added to have a handshake between the two sides of the system. The *Packer* module transforms the data bits of one AXI channel into chunks of 32 bits, AXI channels have different widths in their data bits and, the widest channel is the W channel, with 146 bits. The packer module was designed to receive a vector of 160 bits and to transmit five chunks of 32 bits. It always receives a fixed number of bits in order to

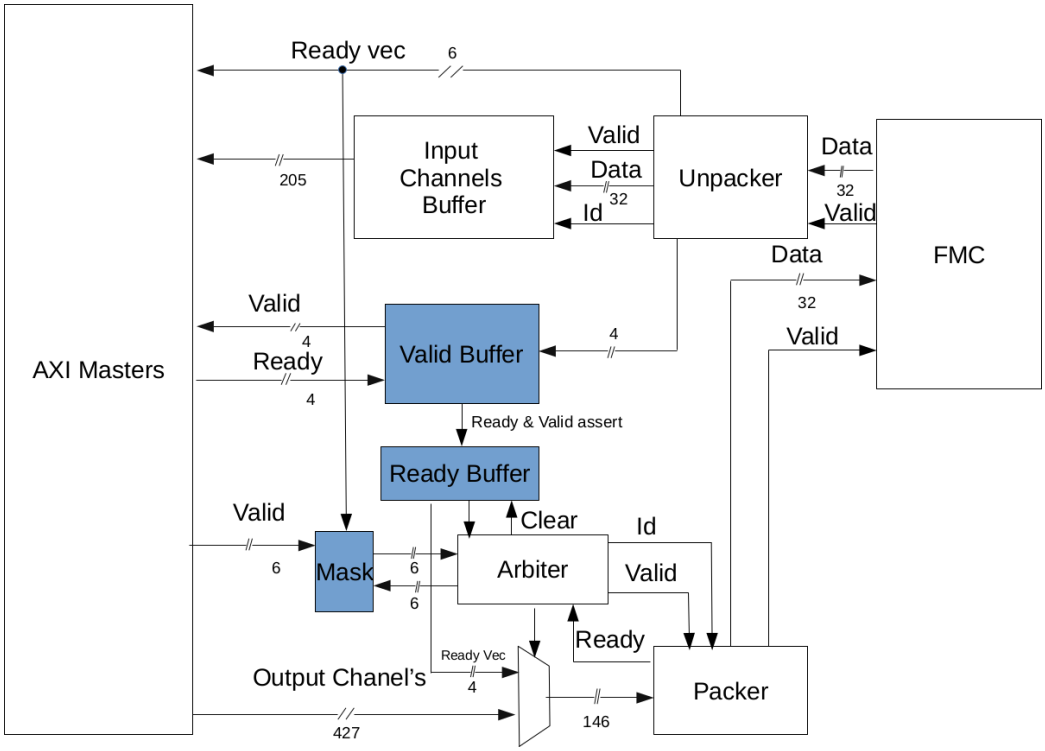

Figure 3.11: Block diagram of the AXI control and the Packer and Unpacker logic.

keep the design as more uncomplicated as possible. The eight most significant bits of this 160-bit vector was used to send an Id number, that ID indicates which channel is being transmitted. Next to the Id, the data bits of a channel is placed. The remaining bits of the vector is filled with zeros. The *Unpacker* module does the inverse transformation of the *Packer* module. It receives five chunks of data of 32 bits and reconstructs them into a vector of 160 bits. When the *Unpacker* has the 160 bits vector, it checks the Id and sends the corresponding bits to the *Input Channels Buffer*. At the same time, it turns on the corresponding *VALID* bit in the *Valid Buffer*. The *Input Channels buffer* is a set of registers that act as buffers for each AXI transaction received, when new data bits of a channel transaction arrives, the data bits will remain in the buffer until the *READY* signal of the channel is received. Since more than one *READY* signal can be asserted in the same cycle, it is possible to set the corresponding bits of each assertion in the *Ready Buffer*. In order to send the *READY* bits that have been asserted from one side to the other side, an extra "channel" was added, called *Ready vec*. This channel has its Id different from the other AXI channels. When the *Unpacker* receives a *Ready*

vec Id, it sends the corresponding *READY* signals to the corresponding AXI channels. In the same cycle, the corresponding bits of the *Mask* module is cleared to allow us to pass another transaction in the corresponding channels.

The last module, the *Arbiter* module, is a logic that determines which channel will be sent to the Packer module, using pair of *READY* and *VALID* signals, the *Arbiter* can determine if the *Packer* can receive a new transaction. When the *Arbiter* selects an AXI channel, it set the bit in the *Mask* module in order to avoid duplicate transactions in the same way as the *Delay Test* system does. In the same cycle, The *Arbiter* activates the corresponding channel in the *Multiplexer*, it sends a complete vector of 160 bits, putting 0 in the bits that are no being used, this vector already includes the channel Id number. When the *Arbiter* selects the *Ready vec* channel, it takes all the bits that are set in the Ready Buffer, sends a clear signal to the *Ready Buffer* and sends the bits to the Packer with the corresponding Id. The Arbiter uses a priority encoder to control the Multiplexer and select the channel that will be sent to the Packer module. This Arbiter follows a fixed priority, giving the highest priority to the *Ready vec*. Then on the Master's side, the next priority is given to the AR petitions, after to the AW channel, and finally to the W Channel. In the Slave, also the highest priority is for the *READY*, then for the R channel, and the last priority is given to the B Channel. This priority order allows that the *READY* and *VALID* Assertions that indicate that a transaction has been completed are transmitted to the other side before starting with a new transaction. Then on the Master's side, the priority goes first to the Reads and after to the Writes. In the Slave's side, the Data reads have more priority that the Write responses.

This *Packetizer* system was tested with an individual test-bench that perform different test cases with different transactions. Then, it was inserted into the system and tested with the ISA test strategy simulating a slower clock in the simulations. Finally, the FPGA test implementation was used to test the system also with a clock divider. This textitPacketizer system has been designed to support the AXI complete of 128 bits data width, and the AXI-Lite of 32 bits that the system has concurrently. In this way, we can use in the FPGA the memory controllers, that are connected to the AXI complete interface, and the FPGA IO controllers, such as UART and SPI, that are connected to the AXI-Lite interface. Also, the option of using internal IO controllers connected to the AXI-Lite interface in the ASIC was added. This feature was connected to an input control pin that controls a set of multiplexers that changes between the internal controllers and the packer-serializer system. Whit the purpose of add this feature, the AXI implementation of the system was modified by adding a second level of combiners, slicers, and crossbars.

I design and implement the modules of this *Packetizer* interface. Nehir Sonmez implemented the asynchronous FIFO queues in Bluespec Compiler, version 2014.07.A. Finally, the design was verified by a test-bench implemented by Alireza Monemi.

3.5.5 Synchronizers and CDC logic

The use of an FMC cable required to have multiple clock frequencies. In order to preserve the integrity of multiple single-ended signals connected through an FMC link connection, it was necessary to slow down the frequency around 25Mhz using two buses of 32 bits; there is one bus per direction. With the purpose of not to slow down the frequency operation of the complete system, it was necessary to use FIFOs to be able to facilitate CDC. In order to reduce the possible issues related to CDC, we used a derived clock for the FMC in our ASIC. The FMC clock is simply the ASIC clock divided by four and is thus aligned with it, reducing clock complexity. The synchronizer logic was implemented using the derived clock from the ASIC. This clock was used as enable signal in both sides of the implementation to read data from the FMC interface. The data will remain on the bus until a new transaction arrives coupled with a new enable signal. The enable signal to allow the data to read for one cycle; in this way, we are sure that the data in the bus is stable when the receptor reads it. In order to avoid metastability problems, the enable signal is synchronized to the receptor domain clock by a couple of registers.

Finally, a synchronization circuitry has been added to the asynchronous reset input to avoid any metastability problems. Among the different alternatives, the most widely used circuit for asynchronous reset synchronization was used (Figure 3.12).

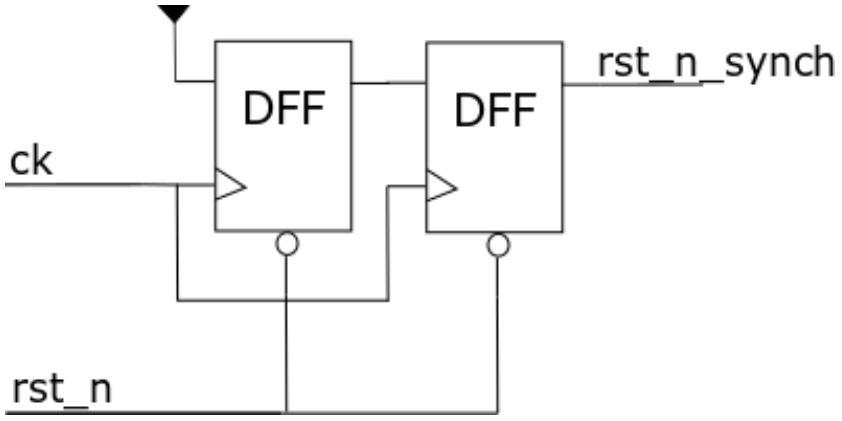

Figure 3.12: Reset synchronizer scheme

3.6 JTAG Debug support

As is mentioned in Section 4.2, torture tests helped to identify errors with the use of simulation tools before uploading our design to the FPGA. Also, the *Boot linux* verification was used to determine that the final RTL implementation of the system was working correctly. However, between these two steps, it is possible to find errors as well. The system might successfully finish all the ISA and torture tests but not booting still, and these errors can be caused not only from incorrect coding but also from missing connections that are different between simulation and FPGA wrappers to interconnect the FPGA IO. On the other hand, after manufacturing the chip, physical errors regarding the memories could be the reason for our SoC to stop functioning correctly. Therefore, we developed a backup-test system which allows us to verify at run-time the state of our SoC, and to be able to inject internal tests without using the *packetizer* interconnection. This system is referred to in RISC-V literature as Debug Ring [35]. *lowRISC* company has developed a Debug Ring infrastructure which includes several characteristics, such as:

- Generate a function trace (enter and leave functions) for the program.
- Allow minimal intrusive software instrumentation with trace events.
- Memory access and initialization.
- Reset the system and cores remotely
- Serial communication (console) via the Debug System.

The main issues with this infrastructure were; first, its code is not FPGA synthesizable from scratch, and *lowRISC* provides a pre-built bitstream instead for testing it; second, this Debug Ring controls the SoC initialization for Linux booting operations, this means that the SoC would not be capable of booting the Linux kernel without the Debug Ring and user interaction.

Due to these reasons, we developed our infrastructure to backup-test our SoC. This infrastructure includes the necessary functionalities to test and validate the operation of our SoC after the FPGA upload and chip manufacturing.

Similarly, as the *lowRISC* debug ring implementation, we are based on the Open SoC Debug library (OSD) [36], which provides a plug and play communication interface with a base architecture and the Generic Logic Interfacing Project (GLIP) [37], which gives a generic data exchange protocol based on FIFO queues. The next functions are included in our current design, some of them taken from the OSD standard, and others included to fulfill internal requirements from the project.

- System control to reset and stall the entire SoC.
- Non-invasive core verification.
- Core register read/write operations.
- Core stall control.
- Threshold operation in the core.
- L2 access and initialization.
- Physical serial communication through JTAG protocol.
- Simulated serial communication through TCP server/client protocol.

The tracer functions were not included mainly because their implementations incur in high area consumption due to the usage of big memory buffers to store core states during the execution.

The threshold operations included will stall the core pipeline when the trigger condition is reached. This condition can be configured for three different sources: PC at the decode stage, PC at commit stage, and last memory accessed address.

Figure 3.13 shows the block diagram from our debug ring implementation inside the FPGA. It is divided by SW (Software) and HW (Hardware). On the software side, the crucial aspects of underlining are the usage of the OSD and GLIP c++ libraries. These controls the communication with the debug interface and allows us to easily connect with different technologies as JTAG, UART-rs232, and TCP without changing the c kernel code used.

In the hardware side (Referring to the ASIC), there are three modules to underline, and these are:

- System Control Manager (SCM): This module controls the reset and stall signals for all the SoC.
- Core Control Manager (CCM): Controls all the interactions with the core as the write/read registers.
- Memory Access Manager (MAM): This can access (read/write) the L2 memory to initialize values or even include standalone tests.

The front-end JTAG interface based on OSD was implemented by Abraham Josafat Ruiz, and the back-en debug ring architecture was implemented by Julian Pavón Rivera.

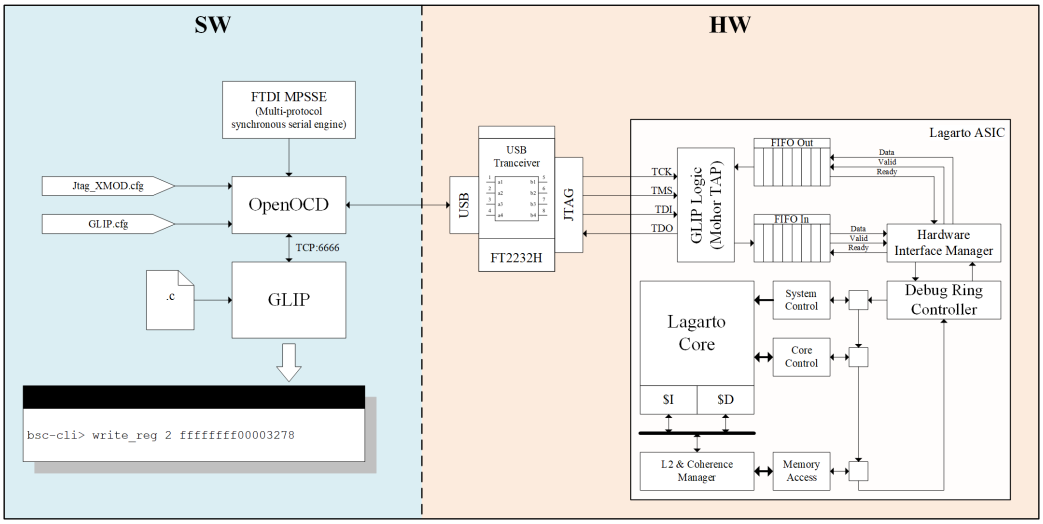

Figure 3.13: Debug Ring Block diagram and interconnection.

3.7 Custom peripheral controllers

3.7.1 PMU

Performance Monitoring Units (PMUs) are fundamental for developers and researchers. It is necessary to measure events that happen inside the processor to understand the effects of applications over the micro-architecture of a given implementation. This measure necessity takes special importance in real-time and critical safety systems where even the commercial implementations of PMUs may produce inaccurate results or not enough granularity to extrapolate strong conclusions. The PMU *preDRAC* implementation focuses on scalability and flexibility. Result of that choice We have decided to implement the PMU as an AXI-Lite peripheral, as can be seen in Figure 3.1. The PMU can be configured at any time by the processor and run independently of the program flow until a threshold is reached or the processor decides to access again.

To maximize flexibility and re-usability, the configuration of this module is parametric. The default parameters of this module can be found in Table 3.3.

The block diagram of the PMU is represented in Figure 3.14. This module interface with the processor through an AXI-Lite interface, it receives events from the processor to monitor and can generate interrupts once a given quota has been reached. Note that as it is implemented now, the interrupts are not connected to the core and will not generate exceptions, but that is something

Table 3.3: Default configuration parameters for the PMU module.

Parameter	**Default Value**
N° Counters	16
Data bus Size (32/64bits)	32
N° Configuration registers	5
Overflow interruption	yes
Quota monitoring interruption	yes

N° counters can be any natural number up to 64, *Data bus* size can be set to 32 or 64 bit, *N° Configuration registers* allow to configure as many entries as needed in to the memory space of the AXI wrapper and can take any natural number up to 256. *Overflow interruption* and *Quota monitoring interruption* are binary fields that allow to instantiate the mechanism to generate each interrupt.

that could be implemented in the future.

The internal structure of the PMU is composed of six main elements. The AXI-Lite slave control machine also called wrapper, a crossbar or multiplexer to select the events, a set of counters, logic to select interruptions, a bank of registers to store results, and configuration, and finally some additional glue logic.

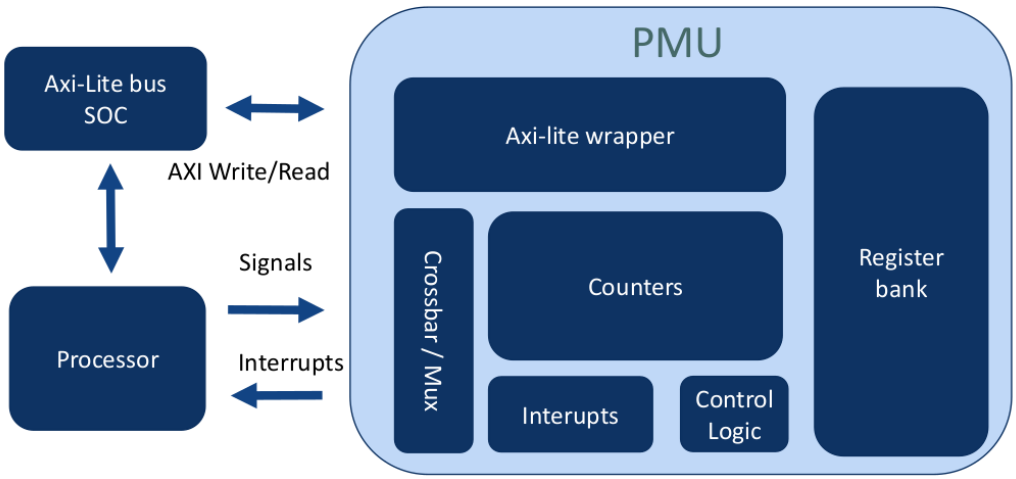

Figure 3.14: Block diagram of the PMU and its functional units.

The wrapper has been connected to a set of registers that are writable and readable. These registers have two functions. One set of registers act as storage for the counters the other work as configuration registers and change the internal state of the PMU operation.

In the submitted design the PMU has been configured with only one configuration register and nine counters that measure:

- Clock cycles
- Instruction cache misses
- Instruction TLB misses
- Data cache misses
- Store instructions
- Load instructions
- Branch mispredictions
- Instruction count

All the interruptions have been disabled.

This module was designed and implemented by Guillem Cabo.

3.7.2 UART

UART is a physical circuit on a micro-controller or a standalone IC, whose primary purpose is to transmit and receive serial data. In the scope of our SoC, the UART module is responsible for handling the communication between the host computer (developer's environment) to the remote machine (Printed Circuit Board (PCB) including SoC and peripherals). The transmitting UART circuit communicates with the respective receiving UART circuit in an asynchronous (no clock signal synchronizes the transmitter's output with the receiver's input sampling) and serial way. The serial property of the UART dictates that the chip transforms the parallel data in its input to a serial packet to be transmitted via its output.

As mentioned above, the UART connection establishes the communication by transforming the data it receives from the data bus to a serial packet, having a specific structure, that it is later transmitted to the receiver end of the communication channel. The general form of the data packet is depicted in Table 3.4.

Table 3.4: The general structure of a UART packet.

Start Bit	**Data Payload Bits**	**Parity Bit**	**Stop Bit**
1 Bit	5 to 9 Bits	1 Bit	1 to 2 Bits

In our case, the serial packet transmitted is designed to consist of 8 data bits, one parity bit (assuming odd parity), and one-stop bit. This configuration gives up a total of 11 bits per transmitted packet. Nevertheless, this is a design choice. The start bit is a signaling level change from logical '1' (which is the default idle state of the UART communication channel) to logical '0'. Next comes the data frame consisting of the total data payload to be transmitted, and right after comes the parity, which describes the evenness or oddness of the previously transmitted number (oddness in our case). Lastly, to signal the end of a data packet, the sending UART drives the transmission line from a low voltage to a high voltage.

The start and stop bits define the start and stop of the serial data packet. When the receiving UART detects a start bit, it starts reading the incoming bits at a mutually, to both the transmitter and the receiver, agreed-upon frequency, which is called baud rate. The latter measures the speed of a data transfer, and it is expressed in bits per second (bps). Each specific baud rate should be respected by both the transmitter *Tx* and the receiver *Rx*, and the accepted deviation can be up to 10% before the sampling fails to recognize the transmitted data correctly. In our design, the baud rate can be configured from the UART SW driver, but the default value is 115200 bps.

In Figure 3.15, we present a high-level overview of the architectural components of the UART design.

The UART module were designed originally for the *Lagarto* SoC v1.0 by Abraham Josafat Ruiz and ported to the *preDRAC* SoC by Vatistas Kostalabros.

3.7.3 SPI

The SD Card Controller is the submodule that is in charge of reading the Berkeley Boot Loader (bbl) and the Linux Operating System (OS) image from the attached SD Card through a series of AXI-Lite read/write transactions and SPI communication protocol packets. The submodule works as an AXI-Lite Slave and is mapped into IO space. It comprises of an AXI-Lite protocol controller and primary Finite State Machine, two FIFO buffers (receiver and transmitter), a central SPI controller (controls the transaction

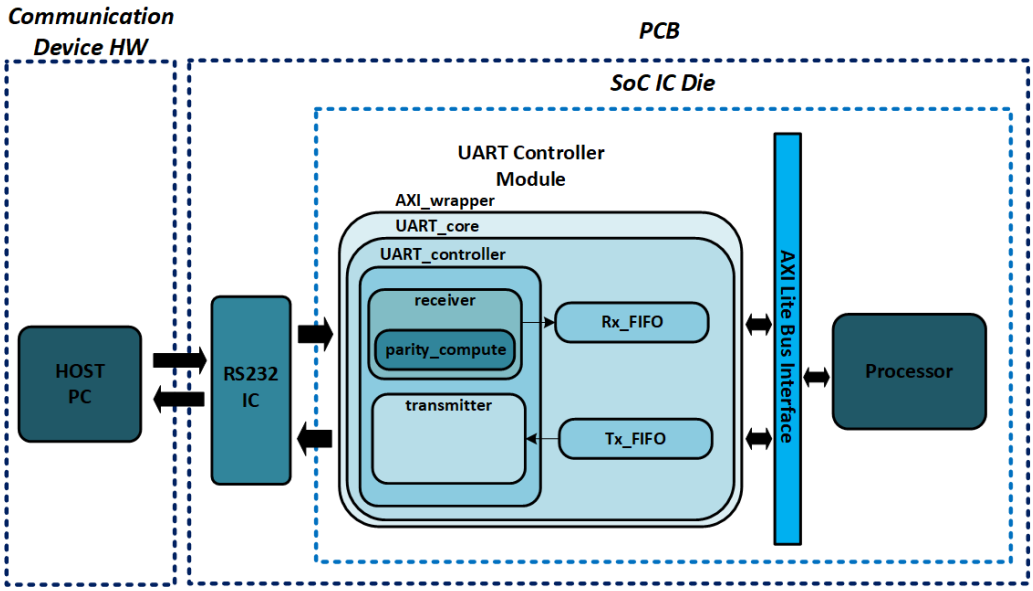

Figure 3.15: High level overview of the architectural modules of the UART controller along with the communication chain components.

between the data exchange mechanism and the corresponding receiver/transmitter FIFO buffers), an 8-bit data exchange mechanism (between mosi and miso) and a ratio-driven clock generator for the SPI clock, Figure 3.16.

SPI Controller

SPI is one of the most used interfaces between a microcontroller and a peripheral IC. It is also one of the two main SD Card communication protocol modes. TheSPI protocol consists of four signals:

- SCLK: SPI clock (generated by the Master)
- CS_N: Chip Select (normally active-low, generated by the Master)
- MOSI: Master Output Slave Input Data (generated by the Slave)
- MISO: Master Input Slave Output Data (generated by the Master)

The SPI controller can act as a Master or a Slave depending on its configuration, if a Master, the SPI controller, will initiate every data transaction between MOSI or MISO, if a Slave, the input and output data will be latched in the positive or negative SPI clock pulse (input).

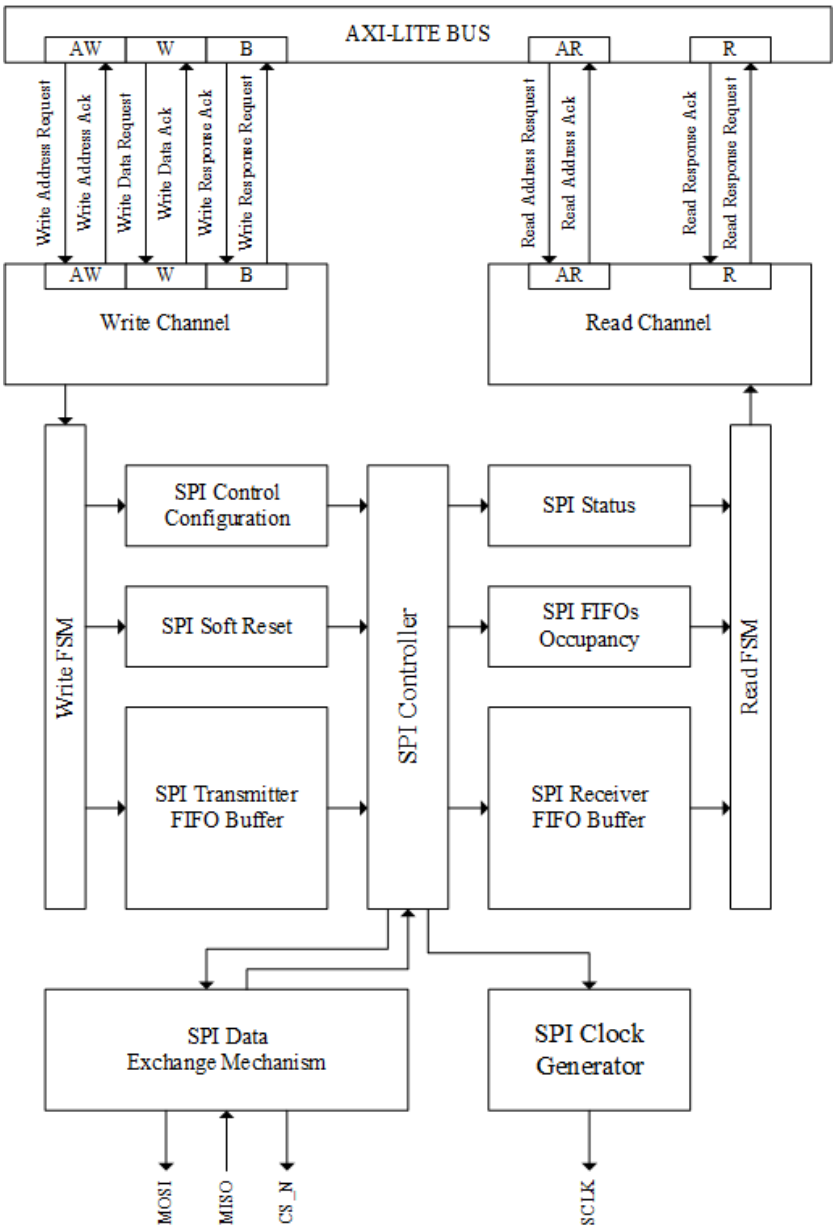

Figure 3.16: AXI4-Lite SPI Slave submodule overview.

The AXI4-Lite SPI Slave module comprises of eleven memorymapped special registers, the complete functionality of the SPI submodule can be controlled by reading and writing into these registers, Table 3.5. These special registers are read and written by the processor and program with the help of the software driver, which is a set of C code functions and address

Table 3.5: AXI4-Lite SPI Registers

Address Offset	**Registers Name**	**Access Type**	**Description**
40h	SPI SR	W	SPI software reset register, writing an 0x0a will cause a soft reset.
60h	SPI CR	R/W	SPI control register, writing to this register will set the SPI configuration, such as SPI mode, FIFOs soft reset and SPI enable.
64h	SPI SR	R	SPI status register, indicates if there some transaction error, as well as the empty and full flags for both FIFOs.
68h	SPI DTR	W	SPI data transmitter register, inserts a new entry to the transmitter FIFO.
6Ch	SPI DRR	R	SPI data receiver register, reads the oldest entry from the receiver FIFO.
70h	SPI SSR	R/W	SPI slave select register, selects the "N" SPI slave.
74h	SPI TX FIFO	R	SPI transmitter FIFO occupancy register, indicates the occupancy of the transmitter FIFO.
78h	SPI RX FIFO	R	SPI receiver FIFO occupancy register, indicates the occupancy of the receiver FIFO.
1Ch	SPI DGIER	R/W	SPI global interrupt enable register (unused).
20h	SPI IPISR	R/W	SPI interrupt status register (unused).
28h	SPI IPIER	R/W	SPI interrupt enable register (unused).

definitions. In order to send a byte from the C code program to the SPI module.

This module was designed and implemented by Abraham Josafat Ruiz.

Chapter 4

Netlist Generation and Verification

4.1 Synthesizable RTL implementation

For generating the output netlist that will be used in the physical level design stages, it was necessary to adjust the RTL sources in order to have a correct output from the synthesis processes according to the following considerations.

- To avoid not synthesizable constructs for the target, e.g., initial statements.
- To ensure that all control registers are properly initialized in a reset statement.
- It is recommendable to use a homogeneous reset strategy for the complete design.
- To avoid unconnected and not used signals in the implementation.
- To avoid width mismatches between signals.
- To avoid undefined outputs for a given combination of inputs in combinational logic.
- To avoid inferred latches constructs.

For the *preDRAC* project, an asynchronous reset was chosen. Asynchronous reset was the more straight forward in order to avoid problems with registers that were not correctly reset. When the synthesis was performed using synchronous reset, the tool does not connect the reset signal to

the register's reset input. Instead, the reset signal was handled as part of the data-path and connected to the D input. In this case, the reset signal has not a priority over the enable signal, causing that, in some cases, the register was not correctly reset. In order to use synchronous reset, it is necessary to add a directive in the RTL code for the synthesis tool [25]. The primary reset signal was implemented as active low, but the negated of this signal was used in some modules.

To verify that the code was synthesizable and the warnings that it could have did not imply an error in the description, a linting process was performed to the code. Genus tool was used to perform the linting. The elaborate command was used for that purpose.

4.2 Verification Strategy

The testing and verification strategy for the DRAC core consisted of 4 different ways:

- RISC-V ISA tests.
- Basic benchmarks.
- Toruture test.
- FPGA test.

4.2.1 RISC-V ISA tests

The *riscv-test* repository is part of the RISC-V SW tools. This repository is a set of open-source assembly programs that test each instruction individually. The RISC-V cross-compiler compiles each test, and all regular assembly directives can be used. The test includes integer, floating-point, and vector extensions; it is possible to use different levels of execution privilege, virtual memory, and the timer interrupt [38]. In our verification pipeline, this set of tests is the first stage that every addition and modification to the SoC should pass in RTL simulation.

4.2.2 Basic benchmarks

The Mälardalen WCET Benchmarks

As an initial effort to evaluate the performance of the Lagarto core, we used the Mälardalen Worst-Case Execution Time (WCET) benchmarks [39].

The Mälardalen WCET benchmarks were collected in 2005 from several researchers within the WCET field and were put together from the WCET group of the respective university in Sweden. The purpose of the Mälardalen WCET benchmarks is to have a standard, easily available set of test programs for WCET methods and tools. All of the benchmarks are available on a web page [40].

In Table 4.1, we present the four benchmarks we decided to port to our infrastructure.

Table 4.1: Mälardalen benchmarks ported to Lagarto core.

Program	**Description**	**Comments**
bsort100	Bubble-sort program.	Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 200 integers.
fibcall	Iterative Fibonacci, used to calculate *fib(30)*.	Parameter-dependent function, single-nested loop.
fdct	Fast Discrete Cosine Transform	Lots of calculations based on integer array elements.
matmult	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.

Even if all the benchmarks can be run without modifications (i.e., the programs contain their own inputs), we had to make some minor modifications to their source code. Those modifications were essential in order to provide an execution time, instructions executed reporting mechanism. The obvious cross-compilation of the benchmarks was performed without major obstacles since the cross-compilation tool-chain of the RISC-V project is at a mature enough stage with support from *gcc* and many libraries available (i.e., UART). To elaborate, we inserted some in-line assembly code before and after the Region Of Interest (ROI). The specific assembly code reads the value of two specific user-space Control Status Registers (CSRs) (one counting clock cycles and the other counting executed instructions). In this way, we are able to subtract the following values and obtain the total number of instructions and cycles the ROI needed to be executed. These changes were necessary because the benchmarks run in the bare-metal mode without any support from an underlying OS (i.e., Linux). The developer can, in a successful run of the benchmarks, observe on his host machine screen the

result of the benchmark itself. (e.g the specific Fibonacci number in case of *fibcall* or the matrix multiplication result in case of *matlmult*) as well as the number of instructions executed along with the clock cycles needed for these instructions to execute.

4.2.3 Torture Test

In order to test the proper operation of the design, we wanted to test the functionalities regarding instructions. To do this, we used the torture tests from *lowRISC*, which allow us to execute them in our RTL design and compare the results with the results from a reference model, in Spike ISS.

The torture test validation consists of extracting the contents of the register file and a portion of the memory of each execution. With this extracted data, the system constructs a signature. We compare later them against the reference model in Spike. These torture tests were vital to perform the functional verification of the design. As they are automatically generated, we could create big sets of tests and cover more possible combinations of instructions.

Randomization

The torture tests are generated randomly using a configuration read from a Makefile. This random generation can be controlled or manipulated by changing the "default.config" file to focus on certain sides of the architecture or to leave specific types of instructions out of the tests, Listing 4.1.

```
1  torture.generator.nseqs      200
2  torture.generator.memsize    1024
3  torture.generator.fprnd      0
4  torture.generator.amo        true
5  torture.generator.mul        true
6  torture.generator.divider    true
7  torture.generator.segment    true
8  torture.generator.loop       true
9  torture.generator.loop_size  64
10
11 torture.generator.mix.xmem     10
12 torture.generator.mix.xbranch  20
13 torture.generator.mix.xalu     50
14 torture.generator.mix.fgen     10
15 torture.generator.mix.fpmem    5
16 torture.generator.mix.fax      3
17 torture.generator.mix.fdiv     2
```

```
18 torture.generator.mix.vec      0
```

Listing 4.1: Diferent parameters for Test generation

With this configuration file, it was also possible to break the sequentiality and change the length of the generated test.

The final objective is to compare the memory signature extracted from our design's execution against a golden model to verify the proper operation of the architecture. Execution

To obtain both signatures, we use different software. On one side, we simulate our design using the Verilator, a cycle-accurate RTL simulator, which runs the RTL behavior of the processor using the torture test as input. As the reference model, we use Spike ISS, which is a RISC-V simulator that executes the instructions in C.

Both these tools allow us, using the "+signature" option, to see the portion of the memory that contains the registers with the results.

Behavior

Another important point is how we get the results of the tests. In this case, the torture tests use memory to deliver the results. These tests have a clear structure in which there are different parts, one initialization load part, the test with the instructions to execute, a group of stores that saves the registers in memory and a final memory section in which the final results will be stored as is shown in Figure 4.1. That way, after the instructions have finished, and we have those results in the registers, a sequence of stores saves them in the data section, which is then extracted at the end, ready to be compared.

4.2.4 FPGA test

To be have a closer implementation of the RTL code, we made use of the FPGA implementation to test the different components of the system. The implemented peripherals permit the validation of the basic functionality of the system. The basic Test flow consists of compile a program, loaded its hexadecimal output into an FPGA bit-stream, and finally, connect the UART to see the correct output. We have a set of basic test that stresses the different parts of the system; Peripherals, DDR3 memory, etcetera. The basic implemented FPGA tests are listed below.

- Hello world.
- UART test.
- SPI SD card test.

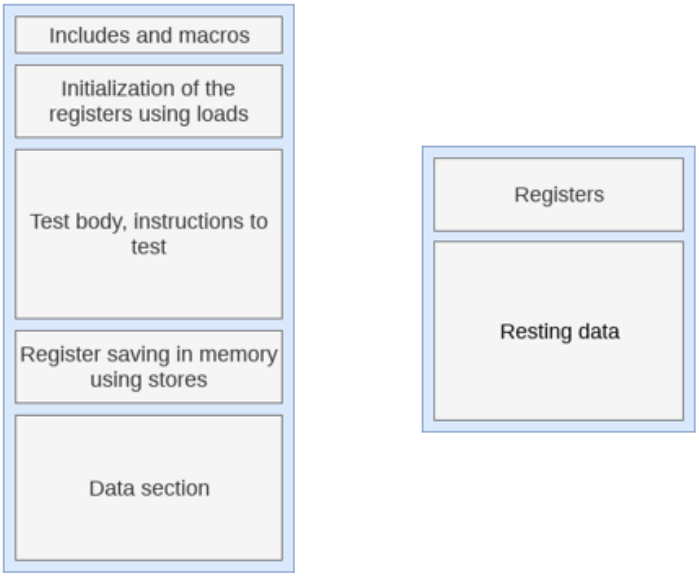

Figure 4.1: The structure of the code of the torture test and the structure of the signature.

- SPI SD card test.
- PMU test.
- Mälardalen WCET Benchmarks
  - bsort100
  - fibcall
  - fdct
  - matmul

4.2.5 RTL simulations

The RTL design simulation was developed in two stages. In the first stage, the Verilator simulator [41] was used in order to verify the correct functionality of the implementation. This stage permit to have a first and fast validation of the added features to the system. With this target in mind, the Verilator tool was selected to develop this stage because of its simulation features. Some features of Verilator are listed below.

- It is an open-source Verilog HDL simulator.

- It compiles synthesizable RTL code plus some SystemVerilog and Synthesis assertions into a C++ or SystemC code.
- It was designed for large projects where fast simulation performance is of primary concern.

As part of the verification strategy, a *Makefile* was implemented. This *Makefile* along with Verilator run a set of *ISA test* that verify that the system executes correctly each one of the different instructions of the implemented *RISC-V* ISA.

The second stage of the RTL simulation implied a more robust simulation tool. It was required to use SRAMs macro-cells libraries, and it was not possible to import and read the models with Verilator. For this reason, the cadence Incisive Enterprise Simulator [42] was used as an alternative. The RTL simulations performed with Verilator were ported to the Incisive simulator, and The same ISA tests methodology was used in order to verify the correct integration of the SRAM macro-cells. This simulator is capable not only to perform RTL simulations but also, it can run post-synthesis and post-PnR GLS using the corresponding netlist.

4.2.6 Gate Level Simulations

Gate-level simulation covers cases that RTL simulation does not take into account. Including the timing simulation and unknowable state signals due to synthesis constructs that behave differently respect to the RTL counterparts.

After a RTL design is synthesized, errors as timing hold violations can be exposed only after the netlist generation. For this reason. gate-level simulation is used to verify these kinds of errors.

Two main inputs are necessary to perform a GLS, first, a netlist, that is a representation of the instances associated with a technology library that are all the gates that form the described system, typically this netlist is the output of the synthesis and is in Verilog format. Second, a timing file that specifies all the delays that are present in the netlist, this file is also associated with the technology library, and it is necessary to perform static timing analysis.

The netlist is the primary output of the synthesis process. It is attached to an SDF file format. This SDF file has the information of the associated timing delays on the instances of the netlist.

GLS for *preDRAC* were performed using the *Incisive Enterprise simulator*, a Tcl script was implemented in order to run the ISA test performed in the RTL simulations. GLS is not only a functional simulation, but it also

simulates the timing of the system. i.e., GLS expose errors in set-up or hold that the data that is driven to the registers could have. In this way, it is important to consider that the IO of the system should not change in perfect synchrony with the clock. It should change with some delay after the edge clock.

4.3 Synthesis

This section describes the set-up used and the results obtained for the synthesis. Synthesis maps the RTL description to a Standard Cell library and generates a gate-level netlist that will be used in the Place & Route phase. The synthesis tool is Cadence Genus version 17.11-s014_1.

4.3.1 Inputs and Directory Structure

The information needed in the synthesis phase is the following:

- RTL files (Verilog, SystemVerilog, or VHDL.)
- Liberty files for standard cells and hard IP blocks. They contain timing and power information.
- LEF files for standard cells and hard IP blocks. They contain physical information as area and pin location. Optional in the synthesis phase, but they improve the quality of the netlist.
- Captable file. It contains information about interconnections parasitics (capacitance and resistance.) Also optional for synthesis, but needed for a better quality of the generated netlist.
- SDC file. It describes the timing constraints: clock domains, clock frequency of each clock, primary inputs, and outputs delay.
- Synthesis scripts. Scripts in Tcl language. Commands for Genus.

The directory structure used for the synthesis is described in Table 4.3.1.

4.3.2 Standard Cell Libraries

There are several standard cell libraries provided by TSMC for the technology TSMC65LP:

Table 4.2: List of directories for Synthesis.

Directory name	Contents
`RTL`	RTL files of the design.
`Constraints`	SDC file.
`RUNDIRGenus`	Tcl files, Makefile and outputs.
`Memories_noBIST_TSMC65LP`	Liberty, LEF and RTL for SRAM memories.
`setup`	sh script to use Genus environment variables.

- Number of tracks: 10 tracks for high speed; 7 tracks for high density.
- Threshold voltage: regular-Vt; high-Vt for low leakage; low-Vt for high speed.
- Timing modeling: NLDM for faster synthesis; ECSM for higher timing accuracy.
- Corner: voltage, process, and temperature conditions.

Libraries are provided for logic cells (*core cells*) and cells for low power techniques (*coarse grain cells*, e.g., level shifters, always-on buffers, etcetera.)

For this project, only core cell libraries are used, 10-track, regular-Vt, and three corners: typical, best case, and worst case with NLDM models. Table 4.3.2 shows the list of standard cell libraries.

Table 4.3: List of Standard Cell libraries.

Corner	Conditions	Library name
Typical	1.2 V and 25C	`tcbn65lphpbwptc.lib`
Fast	1.32 V and 0C	`tcbn65lphpbwpbc.lib`
Slow	1.08 V and 125C	`tcbn65lphpbwpwc.lib`

4.3.3 Hard IP Blocks

The only hard IP blocks used in this project are the SRAM memories needed for the register file, cache memories, and associated tables. Imec provided

these blocks with the needed information for synthesis: Liberty files for different corners, LEF files, and Verilog files (for timing simulation.)

Table 4.3.3 lists the memory blocks used in the design. The same corners as with the Standard Cells, were used for synthesis (Typical, Fast and Slow.)

Table 4.4: List of SRAM cells.

SRAM block name (DxW)	**Description**
`ts6n65lplla32x64m2f`	2 kbits; Integer RegFile banks 0 & 1.
`tsdn65lpa1024x28m4s`	28 kbits; BIPC 0 & 1.
`tsdn65lpa1024x40m4s`	40 kbits; BTB 0 & 1
`tsdn65lpa1024x2m4s`	2 kbits; PHT 0 & 1.
`tsdn65lpa256x128m4f`	32 kbits; L1-I and L1-D (4 instances each.)
`tsdn65lpa4096x128m4s`	512 kbits; L2.
`tsdn65lpa64x80m4f`	5 kbits; L1-I TLB tags.
`tsdn65lpa64x88m4f`	5.5 kbits; L1-D TLB tags.
`tsdn65lpa128x128m4f`	16 kbits; Part of 128x176 L2 TLB tags.
`tsdn65lpa128x48m4f`	6 kbits; Part of 128x176 L2 TLB tags.

4.3.4 Timing Contraints

The timing constraints are specified in the SDC format. In particular, the following information is found in the SDC file:

- Characteristics of the clocks.
- Definition of false paths between independent clock domains.
- Input delay of primary input concerning a reference clock.
- Specified maximum output delay of primary input respect to a reference clock.
- Specified maximum fanout for each gate in the design.
- Specified maximum transition time for each gate in the design.
- Specified capacitance load of the outputs.
- Specified transition time of the inputs.

Table 4.5 shows the commands and values of the SDC file.

Table 4.5: SDC commands and values.

Command	Scope	Value
`create_clock`	clk_core	5 ns period
`create_clock`	clk_JTAG	100 ns period
`set_false_path`	from clk_JTAG	n.a.
`set_input_delay`	[get_clocks clk_core], [all_inputs]	1.25 ns
`set_output_delay`	[get_clocks clk_core], [all_inputs]	0.3 ns
`set_max_fanout`	[current_design]	15
`set_max_transition`	[current_design]	1.2 ns
`set_load`	[all_outputs]	0.5 pF
`set_input_transition`	[all_inputs]	0.2 ns

4.3.5 Synthesis Tcl Scripts

The project is synthesized with a Multi-Mode Multi-Corner flow (MMMC.) While there are no functional modes, the MMMC flow allows having the three corners cases simultaneously. It takes into account to achieve a solution that verifies the timing constraints. The same timing constraints are used for the three corners (see Table 4.3.2.)

The flow is divided into three Tcl scripts:

run_mmmc.tcl Main Tcl file. Calls `run_mmmc_test.tcl` and then performs synthesis, mapping and reports.

run_mmmc_test.tcl Reads technology information (LEF); sets appropriate attributes; calls the appropriate MMMC Tcl file according to the target track library (10 or 7 tracks); reads HDL files; and performs elaboration of the design. It can be used standalone as a fast way to check for inconsistencies in the libraries or RTL.

mmmc10.tcl/mmmc7.tcl Only 10-track `mmmc10.tcl` file was used in this project. Reads timing libraries for different corners; defines corner conditions; reads captable information; defines combined corners of library and captable corners; reads SDC constraint files; defines analysis views.

The analysis views defined in `mmmc10.tcl` are the ones used in timing analysis to calculate the slack. Genus only provides set-up slack, not hold slack. The hold slack is computed in the PnR phase with Innovus. Table 4.6 shows the defined analysis views according to the library and captable corners.

Table 4.6: Analysis views in the synthesis phase.

Analysis view name	Library corner	Captable corner
view_typ	Typical	Default
view_slow	Slow	Default
view_fast	Fast	Default

Table 4.7 shows the most relevant attributes used for synthesis. Finally, Table 4.8 shows the defined path groups for timing analysis.

Table 4.7: Relevant attributes used in synthesis.

Attribute view name	Value	Description
max_cpus_per_server	16	Number of CPUs.
super_thread_servers	batch	Use SGE queue for parallel execution.
hdl_unconnected_input_port_value	0	Value of unconnected input ports.
hdl_undriven_output_port_value	0	Value of undriven output ports.
hdl_undriven_signal_value	0	Value of undriven signals.
syn_generic_effort	medium	Effort of the generic synthesis phase.
syn_map_effort	medium	Effort of the mapping phase.
syn_opt_effort	high	Effort of the optimization phase.

4.3.6 Running synthesis. Makefile

The execution of Genus is called with a simple Makefile to ease the different optional operations. The different defined targets are:

clean Delete all previously generated output files.

test Run elaboration only for the fast identification of problems.

Table 4.8: Path groups for timing analysis.

Path group name	Description
I2C	Input to register.
C2O	Register to output.
C2C	Register to register.
I2O	Input to output.

syn Run full synthesis.

physyn Run synthesis with floorplan information. Not used in this project.

inn Run Innovus with the Stylus interface. Not used in this project.

help Makefile help message.

4.3.7 Synthesis Outputs

This section briefly describes the outputs of the synthesis. They are used in the PnR phase and the gate-level simulation.

The outputs are stored in three different directories inside RUNDIRGenus:

tsmc_10tracks_logs Logfile of the Genus run.

tsmc_10tracks_reports Timing reports; Area reports; Gate count reports; QoR reports; each for the different stages of the synthesis: generic synthesis, mapping, and optimization. Final netlist reports have a name beginning with "final_". Also, the SDC file is re-generated containing gate-level information. Finally, the gate-level design in Genus format is also stored to be re-opened without the need to do the synthesis again.

tsmc_10tracks_outputs Files to be used by Innovus: Gate-level netlist in Verilog format; Innovus Tcl scripts; Genus Tcl scripts; Genus db file. Also, formal verification dofile scripts. Finally, the final SDC and Standard Delay Format SDF file that contains the delay information for each gate for timing-accurate gate-level simulation.

Chapter 5

Results

5.1 RTL simulations

In the Figure 5.1. It can be observed the time digram from the wave forms of the *Packertizer* when an *ISA test* is executed with verilator.

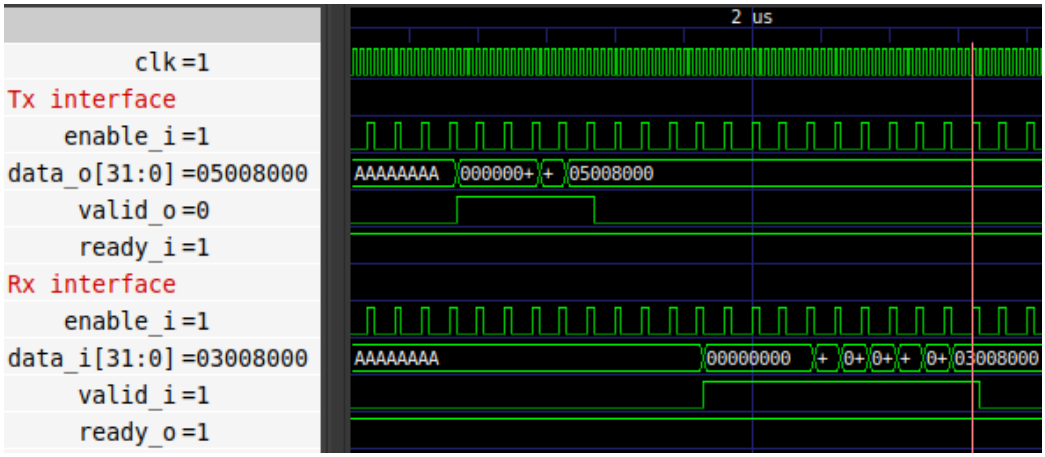

Figure 5.1: Data transaction and its response through the *Packetizer* off-chip interface.

The *AXI* Handshake protocol can be observed in the interface of the master side in Figure 5.2.

In Figure 5.3 can be observed the output given by the Verilator tool when the same *ISA test* test is performed, the messages correspond to the memory transactions achieve by the behavioral model of the simulation.

In Figure 5.4, we can observe the same waveform in the Incisive Simulator.

In Figure 5.5 the log of the memory access in Incisive is displayed. In the case of the RTL simulations performed with verilator, the Makefile Script run

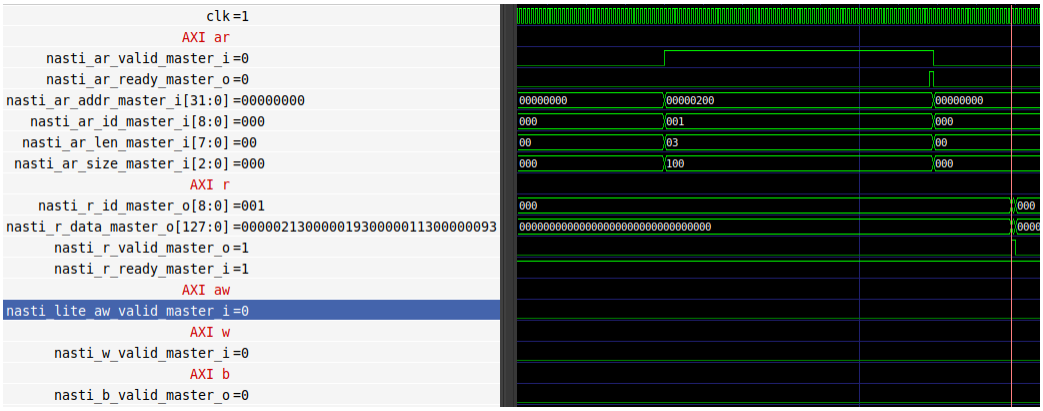

Figure 5.2: Data transaction in the *NASTI* interface of the *Packetizer*.

```
memory read request: 1 @ 200
memory read resp: 1 with data 00000213 00000193 00000113 00000093
memory read resp: 1 with data 00000413 00000393 00000313 00000293
memory read resp: 1 with data 00000613 00000593 00000513 00000493
memory read resp: 1 with data 00000813 00000793 00000713 00000693
memory read request: 1 @ 240
memory read resp: 1 with data 00000a13 00000993 00000913 00000893
memory read resp: 1 with data 00000c13 00000b93 00000b13 00000a93
memory read resp: 1 with data 00000e13 00000d93 00000d13 00000c93
memory read resp: 1 with data f1002573 00000f93 00000f13 00000e93
memory read request: 1 @ 280
memory read resp: 1 with data 0ff0000f 00054863 f0002573 00051063
memory read resp: 1 with data 00000297 00000e13 5480006f 00100e13
memory read resp: 1 with data 1f800293 10129073 00028463 d6428293
memory read resp: 1 with data 34129073 01428293 00000297 3002b073
```

Figure 5.3: Behavioral model of main memory response.

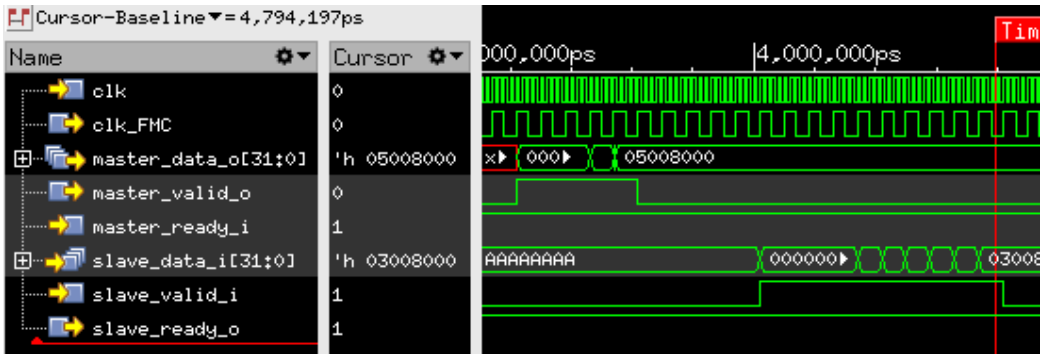

Figure 5.4: Data transaction in the *Packetizer* in Incisive.

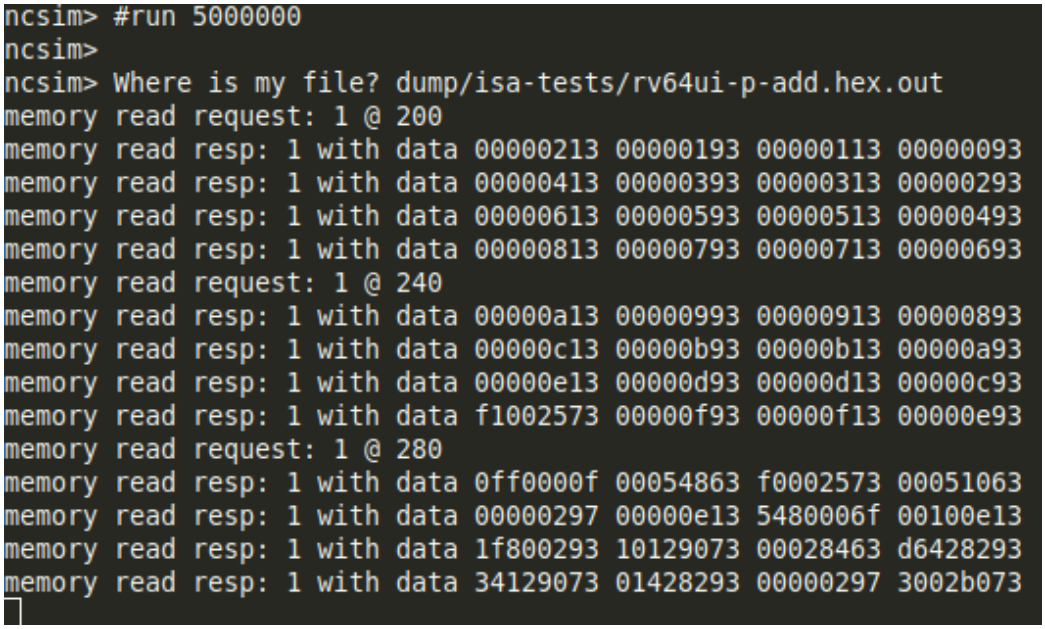

Figure 5.5: Data transaction in the *Packetizer* in Incisive.

automatically checks the correspondign code message that the host interface sends. Thorugh the host Behavioral Model. In the case of the Incisive simulator a TCL script was implemented. By cheking the addresses and the value in the register 1 it is possible to determine if the test was passed or not.

5.2 Synthesis Results

5.2.1 Summary of the results obtained in the synthesis phase

The quality of the synthesis results (QoS) is shown in Fig. 5.6, for each of the three phases in synthesis: generic, mapping, and optimization (final). The relevant results correspond to the column labeled "final".

Of these results, it is worth to highlight that there are no violating paths in terms of timing for the defined timing constraints (200 MHz clock). Table 5.1 shows the single worst case paths for the different analysis views defined in Table 4.6. Note that many other critical paths exist.

There are 88,960 instances. Fig. 5.7 shows a summary of the gates report, where it can be seen that of those 88,960 instances, only 21 correspond to memory cells, which nevertheless represent 79.7% of the cell area and 72.1%

QoS Summary for Top_asic

Metric	generic	map	final
Slack (ps):	150	0	0
R2R (ps):	150	0	0
I2R (ps):	3,260	2,233	1,988
R2O (ps):	4,106	3,657	3,657
I2O (ps):	3,176	2,824	2,824
CG (ps):	no_value	no_value	no_value
TNS (ps):	0	0	0
R2R (ps):	0	0	0
I2R (ps):	0	0	0
R2O (ps):	0	0	0
I2O (ps):	0	0	0
CG (ps):	no_value	no_value	no_value
Failing Paths:	0	0	0
Cell Area:	1,841,871	2,311,766	2,309,746
Total Cell Area:	1,841,871	2,311,766	2,309,746
Leaf Instances:	198,008	86,201	88,960
Total Instances:	198,008	86,201	88,960

Figure 5.6: Relevant data of the quality of synthesis results.

of leakage current.

Type	Instances	Area	Area %	Leakage Power (nW)	Leakage Power %
timing_model	21	1841871.138	79.7	75100.843	72.1
sequential	25664	263667.600	11.4	14051.622	13.5
inverter	4572	6449.200	0.3	680.969	0.7
buffer	6344	11842.400	0.5	1385.291	1.3
logic	52359	185915.200	8.0	12917.438	12.4
physical_cells	0	0.000	0.0	0.000	0.0
total	88960	2309745.538	100.0	104136.163	100.0

Figure 5.7: Number of instances by type and their contribution to area and leakage. Those labeled as `timing_model` refer to macro cells, that is, SRAM blocks.

Finally, Fig. 5.8 shows a summary of the area contribution of the different modules in the design, including the addition of instance area and an estimation of wiring. The total area (Top_asic) is 2,487 mm^2, in line with the initial area budget of 2.5 mm^2 for the core.

Reports generated by Genus include:

- Timing (setup slack) for the different analysis views. The details of the critical path of each analysis view are shown in Table 5.1.

Table 5.1: Critical paths of the different timing views.

View: Slow	
Startpoint:	Rocket/RocketTile/core/INT_REGISTER_FILE_BANK_2_RFArray/CLKR
Endpoint:	Rocket/RocketTile/core/EXE_INTEGER_UNIT/INT_MUL_64B/stg1_result2_q_reg
Parameter	**Value in ps**
Setup	60
Required Time	4940
Path Latency	4940
Slack	0
View: Typical	
Startpoint:	Rocket/RocketTile/core/EXE_INTEGER_UNIT/INT_MUL_64B/stg1_result1_q_reg
Endpoint:	Rocket/RocketTile/core/LATCH_EXE_WB_DATA_TO_CSR_reg
Parameter	**Value in ps**
Setup	26
Required Time	4974
Path Latency	4973
Slack	1
View: Fast	
Startpoint:	Rocket/RocketTile/core/EXE_INTEGER_UNIT/INT_MUL_64B/stg1_result1_q_reg
Endpoint:	Rocket/RocketTile/core/LATCH_EXE_WB_DATA_TO_CSR_reg
Parameter	**Value in ps**
Setup	26
Required Time	4974
Path Latency	4973
Slack	1

- Area report: area occupied by the different hierarchical blocks. The area results are shown in Table 5.2, the Lagarto Core represents the

Instance	Module	Cell Count	Cell Area	Net Area	Total Area
Top_asic		88960	2309745.538	177411.113	2487156.651
Rocket	Top	70962	2202435.538	143355.247	2345790.785
RocketTile	RocketTile	43696	1285223.869	86732.057	1371955.927
dcache	HellaCache	11440	461317.203	22023.374	483340.577
mshrs	MSHRFile	4233	29018.800	7428.636	36447.436
core	LAGARTO_CORE	23173	382511.447	45498.242	428009.689
EXE_INTEGER_UNIT	EXE_INTEGER_UNIT	18207	73123.200	36128.752	109251.952
INT_MUL_64B	INT_MUL_64B	11302	51324.800	22496.150	73820.950
INT_DIV_64B	INT_DIV_64B	1927	8263.200	3768.856	12032.056
csr	CSRFile	2456	12876.000	4593.949	17469.949
bsc_ccm_fifo_out	fifo_glip	598	4181.600	996.122	5177.722
L2HellaCacheBank	L2HellaCacheBank	15057	843925.269	31707.882	875633.151
tshrfile	TSHRFile	11463	50254.000	23936.506	74190.506
wb	L2WritebackUnit	1727	9285.200	3073.415	12358.615
outer_arb	ClientTileLinkIOArbiter	229	690.800	240.397	931.197
bsc_mam_fifo_out	fifo_glip_BUFFER_SIZE32	598	4174.800	996.122	5170.922
pcrControl	PCRControl	3240	18279.600	5972.983	24252.583
conv	NASTIMasterIOTileLinkIOConverter	1216	4804.400	1857.145	6661.545
uart_i_axi_uart_custom_instance_uart_core	avalon_uart_FREQ_CLK50000000_FIFO_SIZE128_DIV_SIZE	6658	36963.600	12730.067	49693.667
packetizer_interface_inst	master_packetizer_interface	3774	22677.600	6595.087	29272.687

Figure 5.8: Area report after synthesis.

17.2% of the design.

- Gates report: statistics on gates used and leakage power.
- Power report.

Table 5.2: Final Area report.

Total Area	
Area	Value in μm^2
Cells	2 309 745.54
Nets	177 411.11
Total	2 487 156.65
Lagarto Core Area	
Area	Value in μm^2
Cells	382 511.45
Nets	45 498.24
Total	428 009.69

The obtained values are important as a feasibility check of the chip. The results will be modified in the final PnR phase, but bad results after the synthesis phase indicate a design problem that should be fixed before starting physical design. These results also provide useful information to designers about the critical points in area and timing that can be improved.

5.3 Gate Level Simulations

Gate level simulations have been performed with Cadence Incisive 15.20-s058. To perform the simulation, Incisive reads the specified netlist and SDF files, generated during synthesis, and annotates the delay information into the design. After annotating the SDF file, as seen in figure 5.9, the signals do not longer change with the clock but are slightly delayed, and some glitches appear.

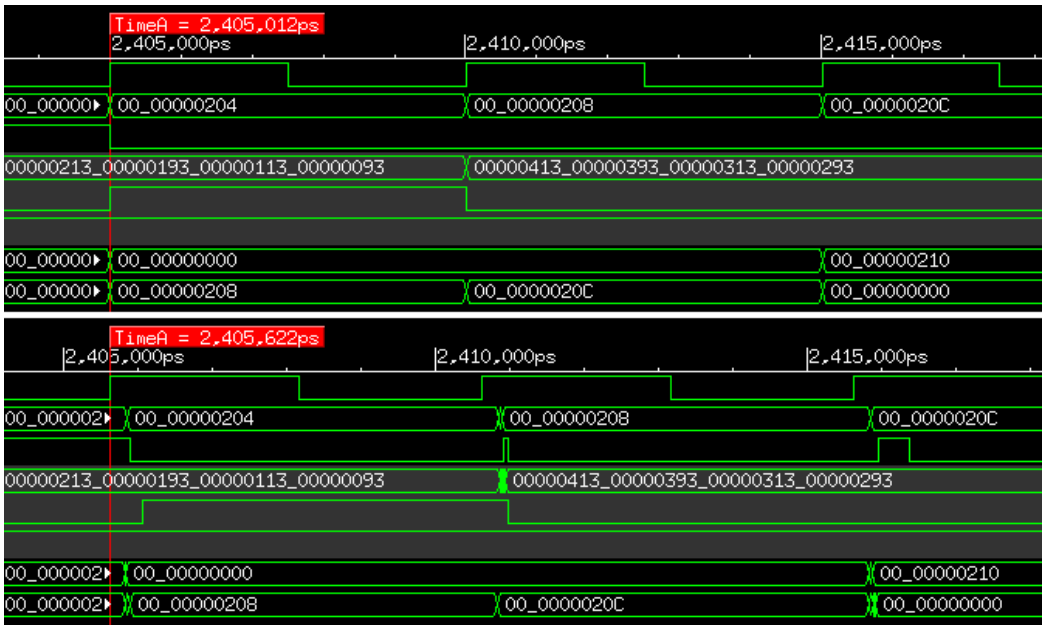

Figure 5.9: Comparison between RTL simulation without delays and gate level simulation with delays

Gate level simulations have been done by executing different *ISA tetsts*. Each test consists of a small program that executes a particular instruction and performs some check to determine if it was correctly executed. Then the result provided by the simulation is a list of the tests performed and if they were successful or not.

The total number of tests, excluding the ones that correspond to the vector and the floating-point extensions, is 395, and all of them passed.

Gate level simulations have also been used to get activity information and perform a more realistic power estimation of the design. Power estimation is done with Cadence Joules v17.11-s003_1. The results of the power estimations performed after the synthesis gate-level are summarized in Table 5.3.

```
Test isa-tests/rv64ui-p-add.hex: Success
Test isa-tests/rv64ui-p-addi.hex: Success
Test isa-tests/rv64ui-p-addiw.hex: Success
Test isa-tests/rv64ui-p-addw.hex: Success
Test isa-tests/rv64ui-p-amoadd_d.hex: Success
Test isa-tests/rv64ui-p-amoadd_w.hex: Success
Test isa-tests/rv64ui-p-amoand_d.hex: Success
Test isa-tests/rv64ui-p-amoand_w.hex: Success
Test isa-tests/rv64ui-p-amomax_d.hex: Success
Test isa-tests/rv64ui-p-amomax_w.hex: Success
Test isa-tests/rv64ui-p-amomaxu_d.hex: Success
Test isa-tests/rv64ui-p-amomaxu_w.hex: Success
Test isa-tests/rv64ui-p-amomin_d.hex: Success
Test isa-tests/rv64ui-p-amomin_w.hex: Success
Test isa-tests/rv64ui-p-amominu_d.hex: Success
Test isa-tests/rv64ui-p-amominu_w.hex: Success
```

Figure 5.10: Extract of 'results.txt' file

The default row refers to an estimation where no switching activity information is provided. The next rows provide the power estimation when using the switching information obtained from a set of *ISA tests* for the following operations:

- add: Addition.
- amoand_d: Atomic memory operation (AMO), which performs logical AND.
- bne: Conditional branch.
- mul: Multiplication.
- ld: Load a value from memory into a register.
- jal: Jump and link. It performs an unconditional jump and stores the address of the instruction following the jump into a register.
- sd: Store a value from a register to memory.

Finally, the idle row contains the power estimation when the processor is in idle state.

5.4 ASIC and PCB deployment

The fabricated chip is show in Figures 5.11 and 5.12.

The chip uses a PCB with the necessary hardware to interconnect the peripherals and a FMC port to interconnect the FPGA that provides the

Table 5.3: Power estimation of the netlist using activity from simulations.

Simulation case	**Leakage** mW	**Internal** mW	**Switching** mW	**Total** mW
default	0.11	190.45	11.00	201.55
add	0.11	138.12	2.58	140.81
amoand	0.11	130.48	2.32	132.91
bne	0.11	138.05	2.45	140.61
mul	0.11	138.02	2.57	140.70
ld	0.11	137.03	2.72	139.85
jal	0.11	138.61	2.32	141.04
sd	0.11	137.98	2.74	140.82
idle	0.11	107.08	2.12	109.30

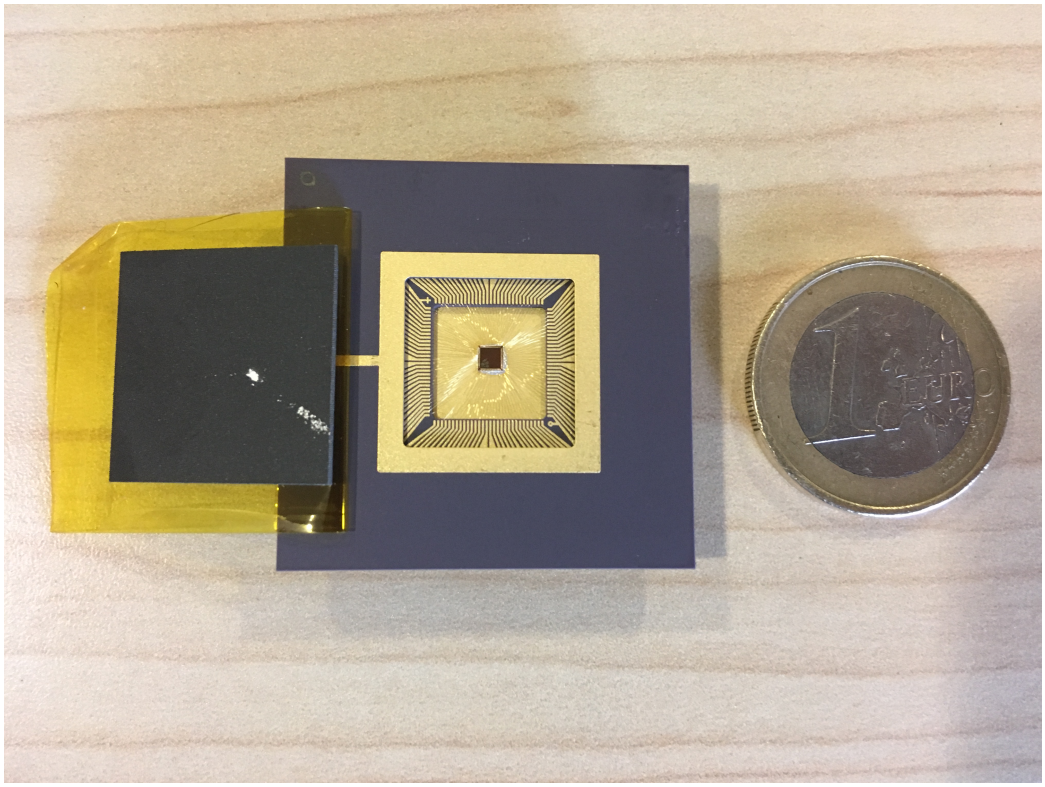

Figure 5.11: *preDRAC* SoC fabricated chip.

main memory access. In Figure 5.13 shows the top view of the PCB and in Figure 5.14 we can observe the bottom view.

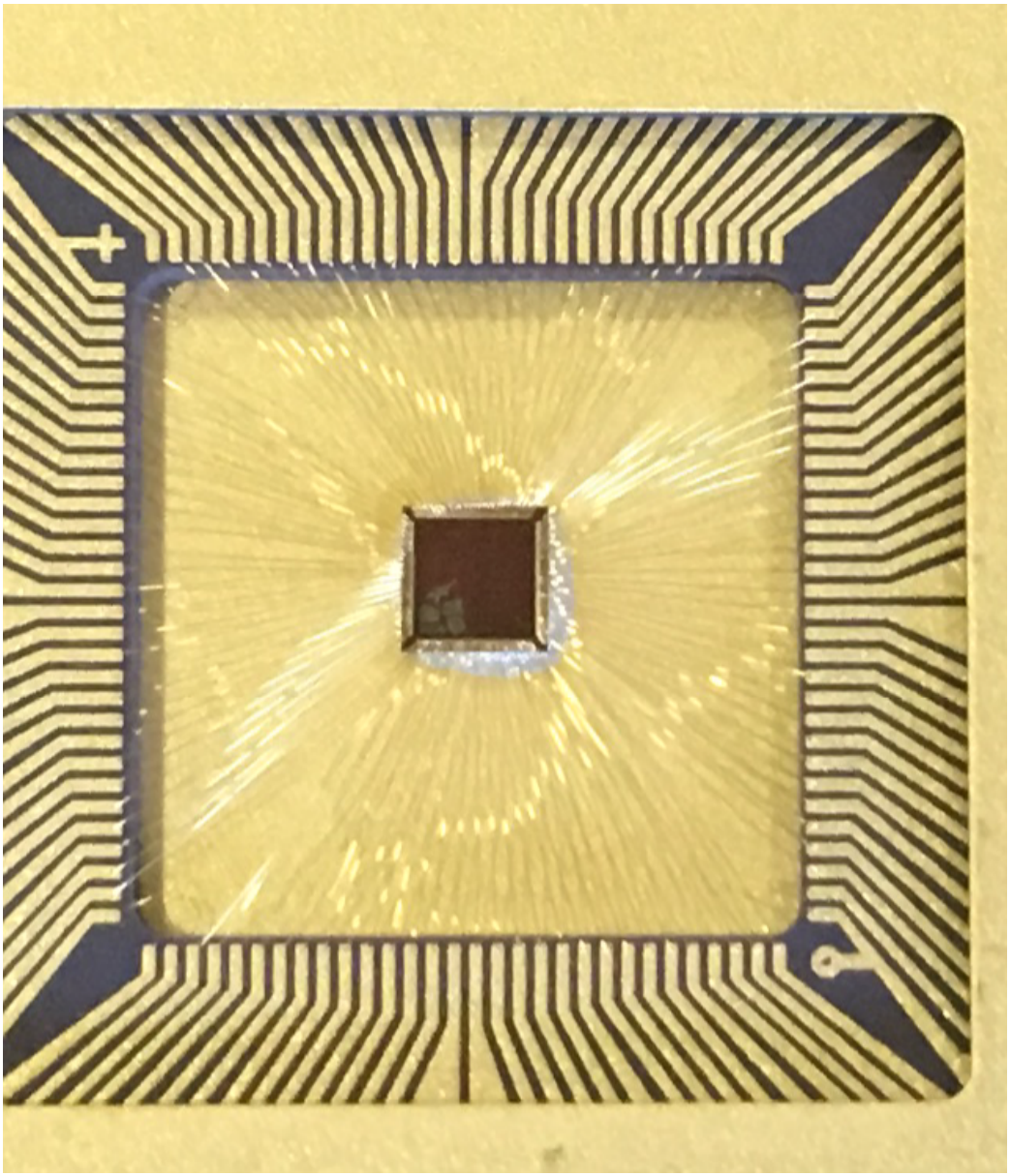

Figure 5.12: Silicon die of the *preDRAC* SoC.

In Figure 5.15 is shown the PCB attached with the FPGA.

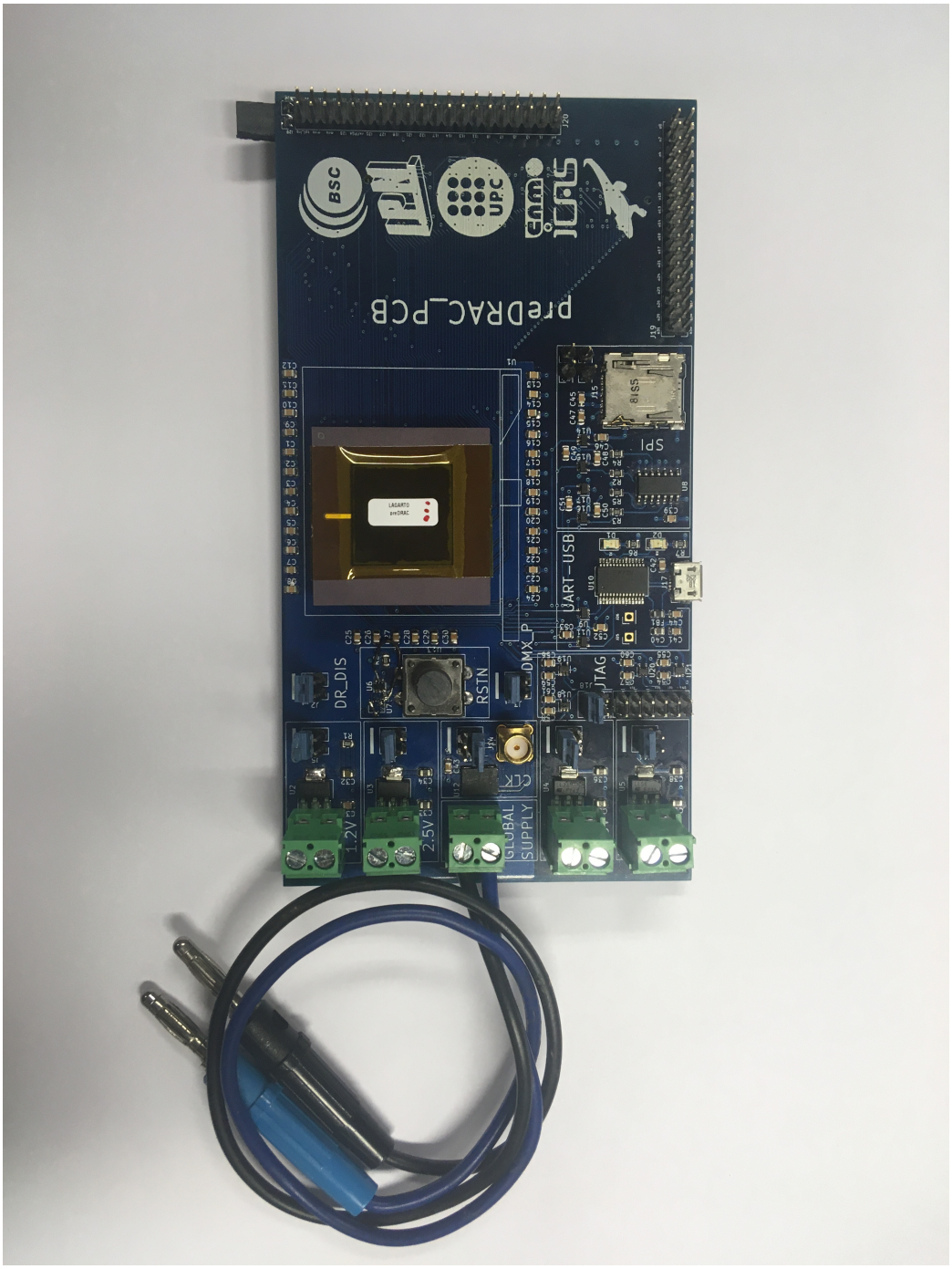

Figure 5.13: Top view of the PCB.

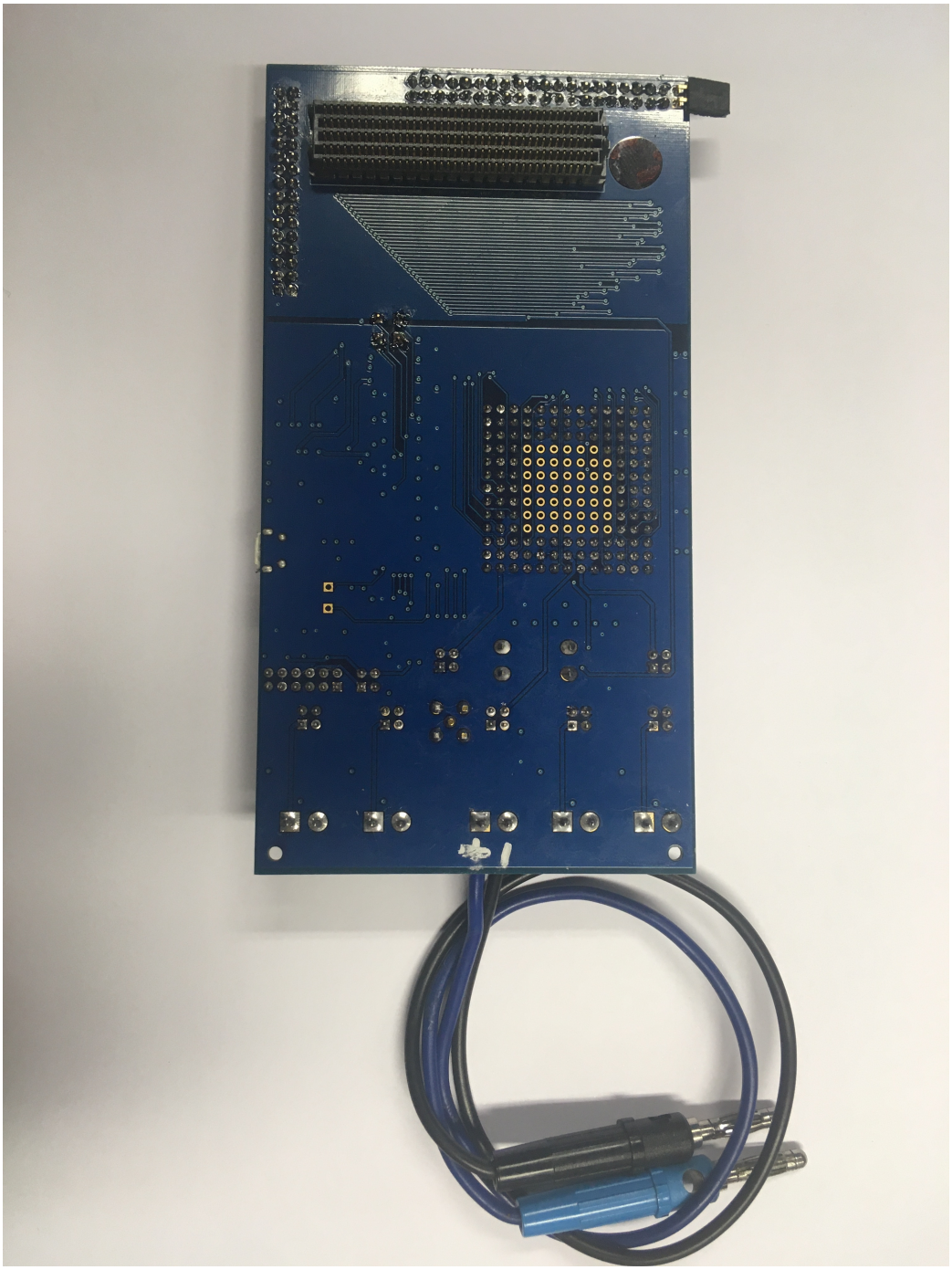

Figure 5.14: Silicon die of the *preDRAC* SoC.

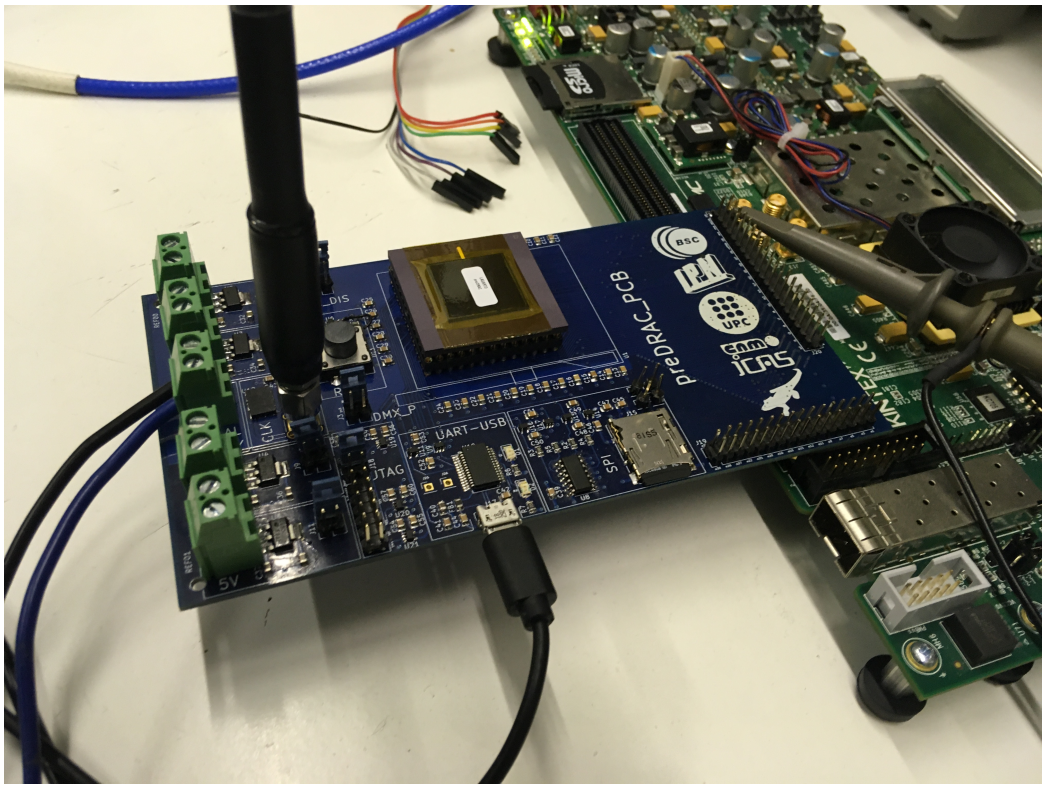

Figure 5.15: *preDRAC* PCB attached to the FPGA.

Chapter 6

Conclusions and Future work

6.1 Overview

The development of the *preDRAC* SoC implied the review of the transition from the RTL design to the physical design. The main challenge was to rethink the design and implementation strategy taking into account the constraints presented when targeting an ASIC deployment. This project implied an experience into technical issues regarding the use of industrial EDA tools. Finally, the product of the project is a good starting point to evaluate the future work that can be done to improve the next iterations of the project and similar projects. The *preDRAC* SoC tape-out and fabrication were an essential step to obtain the necessary experience to define methodologies that permit successful developments in the microprocessor development field.

This thesis work has permitted to document part of these methodologies. Also, It permitted to develop a joint work with a multidisciplinary group to cover several stages and levels in the SoC design and implementation work-flow.

6.2 Conclusions

The design of the modules and features presented in this work were successfully synthesized. The output netlist of this synthesis process was simulated correctly, and the post-synthesis verification tests were passed. Additionally, the further steps of the physical design were performed correctly in this netlist, and the deployed ASIC behave as expected.

We can conclude from these results that the work-flow followed to design, verify and fabricate this ASIC was enough to obtain a functional chip.

The proposed additions to the *preDRAC* SoC including the SRAM macrocells structures, the JTAG debug interface, the peripherals IP controllers, the PMU and the main memory access mechanism were integrated successfully to the SoC.

The implementation of these features was verified with RTL simulations, FPGA prototyping, and GLS. Although we can not unequivocally guarantee that a fabricated ASIC will work free of faults by only following these verification steps, the obtained results show that the system behaves as expected.

The FPGA prototyping of the system was able to test the behavior of the implemented RTL to validate if it generates a system that behaves as expected. This FPGA implementation also permitted to test the fabricated ASIC by interfacing it with the chip to access the main memory system.

The linting procedure using Genus shows the unusable constructs that were not able to be synthesized. Also, most of the warnings that include width miss-matches, dangling logic, unconnected input pins, and unused register were cleaned form in the code. After performing these changes, the RTL verification strategy was rerun. Once the code was cleaned and verified again, the synthesis process was performed.

The GLS shown that the first iterations of the synthesis were not behaving as expected. This bad behavior happened due to many factors. The first one was that the SRAM macro cells should be initialized. In order to not lose generality in the simulations, the SRAM were initialized with random values. The second reason was that some registers had not been reset correctly once the synthesis implements the library constructs. These issues were solved by refactoring the code to use asynchronous resets in those registers.

Finally, the ASIC test methodology shown that the chip can communicate to the main memory through the main memory in the same way as the FPGA implementation does and the JTAG interface can execute instructions and read data as were expected.

6.3 Future Work

As was mentioned in Section 1.1, *DRAC* will have more tape-out rounds. For the next rounds, the objectives will be focused on the increment in performance by improving the core interface to the memory hierarchy. Also, it is planned to integrate a memory controller to substitute the *Packetizer* communication to reduce the high memory latency penalties due to the slow frequency in the physical interconnection and the overhead of the communication protocol. It is planned to add a vector accelerator to evaluate it in an actual ASIC deployment. Also, the necessary hardware modifications

are being developed to use the most recent RISC-V software stack. Finally, DFT features are planned to provide better post-silicon debugging features and to have more coverage in the validation of the correct functionality of the fabricated chip.

Acronyms

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application Specific Integrated Circuit

ATPG Automatic Test Pattern Generation

bbl Berkeley Boot Loader

BSC Barcelona Supercomputing Center

BIST Built-In Self-Test

bps bits per second

CDC Clock Domain Crossing

CIC-IPN Centro de Investigación en Computación del Instituto Politécnico Nacional

CMOS Complementary Metal-Oxide-Semiconductor

CNM Centre Nacional de Microelectrònica

CSR Control Status Register

DDR Double Data Rate

DFT Design For Testability

DRAC Designing RISC-V based Accelerators for next generation Computers

DSP Digital Signal Processing

DUT Device Under Test

EDA Electronic Design Automation

EPI European Processor Initiative

ESL Electronic System Level

FIFO First In, First Out

FPGA Field Programmable Gate Array

GDSII Graphics Data System II

GLIP Generic Logic Interfacing Project

GLS Gate Level Simulation

HBWIF High BandWidth InterFace

HDL Hardware Description Language

HPC High Performance Computing

HW Hardware

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IO Input-Output

IoT Internet of Things

IP semiconductor Intellectual Property

ISA Instruction Set Architecture

JTAG Joint Test Action Group

L1-D Level 1 Data cache

L1-I Level 1 Instruction cache

L2 Level 2 Cache

MMU Memory Management Unit

OS Operating System

OSD Open SoC Debug library

PCB Printed Circuit Board

PLD Programmable Logic Device

PLB Processor Local Bus

PLL Phase-Locked Loop

PMU Performance Monitoring Unit

PnR Place and Route

PULP Parallel Ultralow-Power Platform

RAM Random-Access Memory

ROI Region Of Interest

RTL Register Transfer Level

SCM Standard-Cell Memory cut

SD Secure Digital

SDRAM Synchronous Dynamic Random-Access Memory

SD Secure Data

SerDes Serializer/Deserializer

SoC System on Chip

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

SW Software

TLB Translation Look-aside Buffer

UART Universal Asynchronous Receiver-Transmitter

ULP UltraLow-Power

UPC Universitat Politècnica de Catalunya

VLSI Very-Large-Scale Integration

References

[1] C. Celio, P. Chiu, K. Asanović, B. Nikolić, and D. Patterson, "Broom: An open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos," *IEEE Micro*, vol. 39, pp. 52–60, March 2019.

[2] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–12, 2019.

[3] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8, Sep. 2017.

[4] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP '17, (New York, NY, USA), pp. 2:1–2:8, ACM, 2017.

[5] E. Commission, "European processor initiative: consortium to develop europe's microprocessors for future supercomputers." https://ec.europa.eu/digital-single-market/en/news/european-processor-initiative-consortium-develop-europes-microprocessors-future-supercomputers, 2018. [Online; accessed 27-August-2019].

[6] C. Ramírez, C. Hernández, C. Rojas, G. Mondragón, , L. A. Villa, and M. A. Ramírez, "Lagarto i – una plataforma hardware/software de arquitectura de computadoras para la academia e investigación," *Research in Computing Science*, vol. 137, pp. 19–28, 10 2017.

[7] W. Song, "Untethered lowrisc tutorial." https://www.lowrisc.org/docs/untether-v0.2/, 2015. [Online; accessed 31-August-2019].

[8] R. R. Schaller, "Moore's law: past, present, and future," *IEEE SPECTRUM*, pp. 52–59, 6 1997.

[9] J. Shalf, "Hpc interconnects at the end of moore's law," in *Optical Fiber Communication Conference (OFC) 2019*, p. Th3A.1, Optical Society of America, 2019.

[10] B. Martinez, M. Montón, I. Vilajosana, and J. D. Prades, "The power of models: Modeling power consumption for iot devices," *IEEE Sensors Journal*, vol. 15, pp. 5777–5789, Oct 2015.

[11] A. Cheikh, G. Cerutti, A. Mastrandrea, F. Menichelli, and M. Olivieri, "The microarchitecture of a multi-threaded risc-v compliant processing core family for iot end-nodes," in *Applications in Electronics Pervading Industry, Environment and Society* (A. De Gloria, ed.), (Cham), pp. 89–97, Springer International Publishing, 2019.

[12] A. Waterman, Y. Lee, , K. Asanovic, and S. Inc., *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. CS Division, EECS Department, University of California, Berkeley, 20190608-base-ratified ed., 6 2019.

[13] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 2700–2713, Oct 2017.

[14] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pp. 1–3, Oct 2018.

[15] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Burlington: Morgan Kaufmann Publishers, 2009.

[16] D. MacMillen, R. Camposano, D. Hill, and T. W. Williams, "An industrial view of electronic design automation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1428–1448, Dec 2000.

[17] D. Jansen, *The Electronic Design Automation Handbook*. New York: Springer Science+Business Media, 2003.

[18] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, pp. 8–17, July 2009.

[19] R. S. Nikhil, *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*, pp. 129–146. Dordrecht: Springer Netherlands, 2008.

[20] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, April 2006.

[21] "Iec/ieee international standard - behavioural languages - part 1-1: Vhdl language reference manual," *IEC 61691-1-1:2011(E) IEEE Std 1076-2008*, pp. 1–648, May 2011.

[22] A. Martínez, *Generating Optimal Funtional Coverages in Digital Systems*. PhD thesis, Centro de Investigación en Computación del Instituto Politécnico Nacional, Mexico City, 1 2016.

[23] M. Meghelli, S. Rylov, J. Bulzacchelli, W. Rhee, A. Rylyakov, H. Ainspan, B. Parker, M. Beakes, A. Chung, T. Beukema, P. Pepeljugoski, L. Shan, Y. Kwark, S. Gowda, and D. Friedman, "A 10gb/s 5-tap-dfe/4-tap-ffe transceiver in 90nm cmos," in *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pp. 213–222, Feb 2006.

[24] R. Fung, R. Senthinathan, and N. Chan, "Intra-pair differential skew compensation method and apparatus for high-speed cable data transmission systems," U.S. Patent 7 493 509 B2, Feb. 2009.

[25] C. E. Cummings and D. Mills, "Synchronous resets? asynchronous resets? i am so confused! how will i ever know which to use?," in *SNUG (Synopsys Users Group) San Jose, 2002 User Papers*, 2002.

[26] S. Tam, J. Leung, R. Limaye, S. Choy, S. Vora, and M. Adachi, "Clock generation and distribution of a dual-core xeon processor with 16mb l3 cache," in *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pp. 1512–1521, Feb 2006.

[27] ARM, *AMBA AXI Protocol Specification*. ARM, Cambridge, v1.0 ed., 2004.

[28] C. S. Corp., "Hyperram memory." https://www.cypress.com/products/hyperram-memory. [Online; accessed 15-October-2019].

[29] J. Wright, "Design of a lightweight serial link generator for test chips," Master's thesis, EECS Department, University of California, Berkeley, Dec 2017.

[30] Actel, *Implementing an 8b/10b Encoder/Decoder for Gigabit Ethernet in the Actel SX FPGA Family*. Actel Corporation, Sunnyvale, 10 1998.

[31] U. B. A. Research, "High BandWidth InterFace (HBWIF)." https://github.com/ucb-bar/chisel-awl, 2019. [Online; accessed 30-July-2019].

[32] W. W. Terpstra, "Tilelink: A free and open-source, high-performance scalable cache-coherent fabric designed for risc-v." https://content.riscv.org/wp-content/uploads/2017/12/Wed-1154-TileLink-Wesley-Terpstra.pdf, 2017. [Online; accessed 17-October-2019].

[33] XILINX, *kc705 Evaluation Board for the Kintex-7 FPGA User Guide*. XILINX, San Jose, v1.8.1 ed., 7 2018.

[34] R. K. Boddu and P. Kumbhare, *AXI Chip2Chip Reference Design for Real-Time Video Application*. XILINX, San Jose, v3.0 ed., 7 2014.

[35] S. Wallentowitz and W. Song, "Tutorial for the debug preview of lowrisc." https://www.lowrisc.org/docs/debug-v0.3/, 2016. [Online]. Available at: https://www.lowrisc.org/docs/debug-v0.3/ [2019-10-9].

[36] T. O. S. D. Contributors, "The open soc debug documentation library." https://opensocdebug.readthedocs.io/en/latest/index.html#, 2018. [Online; accessed 17-October-2019].

[37] I. for Integrated Systems (LIS), "The open soc debug documentation library." https://www.glip.io/index.html, 2017. [Online; accessed 17-October-2019].

[38] R.-V. Foundation, "riscv-tests repository." https://github.com/riscv/riscv-tests/tree/79064081503b53fdb44094e32ff54a3ab20a9bf2. [Online; accessed 17-October-2019].

[39] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," vol. 15, pp. 136–146, 01 2010.

[40] J. Gustafsson, "Mälardalen WCET benchmarks homepage." https://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2013. [Online; accessed 9-Octuber-2019].

[41] W. Snyder, "Introduction to verilator." https://www.veripool.org/wiki/verilator, 2019. [Online; accessed 2-October-2019].

[42] C. D. Systems, "Incisive enterprise simulator." https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/incisive-enterprise-simulator-ds.pdf, 2011. [Online; accessed 10-October-2019].